**PCT** WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 12/14, 13/00, 15/00

A1

(11) International Publication Number: **WO 00/16200**

(43) International Publication Date: 23 March 2000 (23.03.00)

(21) International Application Number: PCT/IL98/00443

(22) International Filing Date: 10 September 1998 (10.09.98)

(71) Applicant: PERFECTO TECHNOLOGIES LTD. [IL/IL]; Medinat Hayehudim Street 103, 46733 Herzliya (IL).

(72) Inventors: RESHEF, Eran; Lotem Street 16, 85338 Lehavim (IL). RAANAN, Gil; Hadarim Street 19, 42823 Zoran (IL). SOLAN, Eilon; Ha-Yarden Street 5, 46377 Herzelia (IL).

(74) Agent: SELIGSOHN & GABRIELI; P.O. Box 1426, 61013 Tel Aviv (IL).

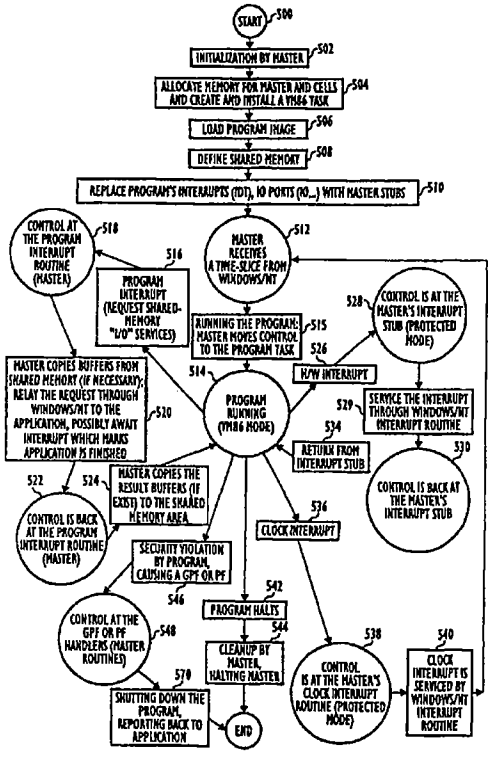
(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

**Published**
*With international search report.*

(54) Title: METHOD AND SYSTEM FOR MAINTAINING RESTRICTED OPERATING ENVIRONMENTS FOR APPLICATION PROGRAMS OR OPERATING SYSTEMS

(57) Abstract

A method for protecting an operating environment on a processor from a rogue program operating on the processor comprising isolating simultaneously executing programs or operating systems is disclosed (502). Memory space for use only by the first program (306) while the first program executing is allocated (504). Communication between the first program and the computer's operating environment is accomplished through a single link employing one of several methods including using shared memory space (508), a dedicated interrupt or a dedicated I/O port (510). The monitor manages (546) a restricted operating environment for the first program (306) on the processor, the restricted operating environment preventing the first program from accessing resources on the processor except for the allocated memory space and the single communication link (516–522).



*FOR THE PURPOSES OF INFORMATION ONLY*

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **AL** | Albania | **ES** | Spain | **LS** | Lesotho | **SI** | Slovenia |
| **AM** | Armenia | **FI** | Finland | **LT** | Lithuania | **SK** | Slovakia |
| **AT** | Austria | **FR** | France | **LU** | Luxembourg | **SN** | Senegal |
| **AU** | Australia | **GA** | Gabon | **LV** | Latvia | **SZ** | Swaziland |
| **AZ** | Azerbaijan | **GB** | United Kingdom | **MC** | Monaco | **TD** | Chad |
| **BA** | Bosnia and Herzegovina | **GE** | Georgia | **MD** | Republic of Moldova | **TG** | Togo |
| **BB** | Barbados | **GH** | Ghana | **MG** | Madagascar | **TJ** | Tajikistan |
| **BE** | Belgium | **GN** | Guinea | **MK** | The former Yugoslav Republic of Macedonia | **TM** | Turkmenistan |
| **BF** | Burkina Faso | **GR** | Greece | | | **TR** | Turkey |
| **BG** | Bulgaria | **HU** | Hungary | **ML** | Mali | **TT** | Trinidad and Tobago |
| **BJ** | Benin | **IE** | Ireland | **MN** | Mongolia | **UA** | Ukraine |
| **BR** | Brazil | **IL** | Israel | **MR** | Mauritania | **UG** | Uganda |
| **BY** | Belarus | **IS** | Iceland | **MW** | Malawi | **US** | United States of America |
| **CA** | Canada | **IT** | Italy | **MX** | Mexico | **UZ** | Uzbekistan |
| **CF** | Central African Republic | **JP** | Japan | **NE** | Niger | **VN** | Viet Nam |
| **CG** | Congo | **KE** | Kenya | **NL** | Netherlands | **YU** | Yugoslavia |
| **CH** | Switzerland | **KG** | Kyrgyzstan | **NO** | Norway | **ZW** | Zimbabwe |
| **CI** | Côte d'Ivoire | **KP** | Democratic People's Republic of Korea | **NZ** | New Zealand | | |
| **CM** | Cameroon | | | **PL** | Poland | | |
| **CN** | China | **KR** | Republic of Korea | **PT** | Portugal | | |
| **CU** | Cuba | **KZ** | Kazakstan | **RO** | Romania | | |
| **CZ** | Czech Republic | **LC** | Saint Lucia | **RU** | Russian Federation | | |
| **DE** | Germany | **LI** | Liechtenstein | **SD** | Sudan | | |
| **DK** | Denmark | **LK** | Sri Lanka | **SE** | Sweden | | |
| **EE** | Estonia | **LR** | Liberia | **SG** | Singapore | | |

# METHOD AND SYSTEM FOR MAINTAINING RESTRICTED OPERATING ENVIRONMENTS FOR APPLICATION PROGRAMS OR OPERATING SYSTEMS

## BACKGROUND OF THE INVENTION

The present invention relates to computer program isolation methods. Specifically, the invention is a method and system for establishing and maintaining a restricted operating environment for a computer program to prevent the program from exploiting bugs and/or data of another computer program which shares the same hardware, while at the same time allowing some form of restricted, well-controlled communication

Contemporary computers rely on a special set of instructions which define an operating system (O/S) in order to provide an interface for computer programs and computer components such as the computer's memory and central processing unit (CPU). Current operating systems have multi-tasking capability which allows computer programs to run simultaneously, each program not having to wait for termination of another in order to execute instructions. Multi-tasking O/S's allow programs to execute simultaneously by allowing programs to share resources with other programs. For example, an operating system running multiple programs executing at the same time allows the programs to share the computer's CPU time. Programs which run on the same system, even if not simultaneously with other programs, share space on the same permanent storage medium. Programs which are executing simultaneously are presently able to place binaries and data in the same physical memory at the same time, limited to a certain degree by the O/S restrictions and policy, to the extent that these are properly implemented. Memory segments are shared by programs being serviced by

the O/S, in the same manner. O/S resources, such as threads, process tables and memory segments, are shared by programs executing simultaneously as well.

While allowing programs to share resources has many benefits, there are resulting security related ramifications. Some programs can have problems in execution due to mistakes or bugs in the program's instructions, or from conflicts with other running programs, or from ill-formatted or mischievous input handed to them. Further, some programs have been circulated which have intentionally embedded mistakes in them so that the program runs astray and becomes a so-called computer virus. Whether by mistake, or by a programmer's malicious intent, many major problems can, and often do occur, which can be traced back to these untrusted programs whose execution results are unpredictable and undesired. These problems include allowing the untrusted or rogue program: to capitalize CPU time, leaving other programs without CPU time; to read, forge, write, delete or otherwise corrupt files created by other programs; to read, forge, write, delete or otherwise corrupt executable files of other programs; and to read and write memory locations used by other programs to thus corrupt execution of those programs. As above, this may be a result of an intentionally malicious code, or a bug in an innocent code, or bad input handed to a code, or malicious input handed to a code, or a combination of these.

An example of such a scenario, where a set of trusted program has to run concurrently with a second set of untrusted programs, is a computer connected to the Internet. In this case, the computer may run an O/S, with several user applications, together comprising the trusted set of programs, concurrently with an Internet browser, possibly requiring also the execution of downloaded code, such as Java applets, or

EXE/COM executables, the latter programs comprising the untrusted set. Sometimes the origin of such program cannot be verified, therefore it may be suspected of being malicious; the browser, when browsing an ill-formatted web-site or a malicious web-site may be subject to inputs that attempt to corrupt its behavior, e.g. too large input streams causing buffer overflow and possible undesired execution of code. It is desired that the execution of the untrusted programs has the least effect on the trusted programs, and this effect should be controlled and confined to a restricted form through, for example, preset file or memory locations, specific interrupts, etc.

Many security features and products are being built by software manufacturers and by O/S programmers to prevent such breaches from taking place, and to ensure the correct level of isolation between programs. Among these are generic architectonic solutions such as rings-of-protection in which different trust levels are assigned to memory portions and tasks, paging which includes mapping of logical memory into physical portions or pages, allowing different tasks to have different mapping, with the pages having different trust levels, and segmentation which involves mapping logical memory into logical portions or segments, each segment having its own trust level wherein each task may reference a different set of segments. Since the sharing capabilities using traditional operating systems are extensive, so are the security features. However, the more complex the security mechanism is, the easier it is for a rogue program to bypass the security and to corrupt other programs or the operating system itself, sometimes using these very features that allow sharing and communication between programs to do so.

Further, regarding rogue or virus programs, for virtually every software security mechanism, a programmer has found a way to subvert, or hack around, the security system, allowing a rogue program to cause harm to other programs in the shared environment. This includes every operating system and even the Java language, which was designed to create a standard interface, or sandbox, for Internet downloadable programs or applets.

The vulnerability of computer programs lies in the architecture of the computer operating system itself. A typical prior art operating system scheme is depicted in Fig. 1. The traditional, multi-tasking O/S environment includes an O/S kernel 100 loaded in the computer random access memory (RAM) at start-up of the computer. The O/S kernel 100 is a minimal set of instructions which loads and off-loads resources and resource vectors into RAM as called upon by individual programs executing on the computer, generally indicated at 102. Sometimes, when two or more executing programs require the same resource, such as printer output, O/S kernel 100 leaves the resource loaded in RAM until all programs have finished with that resource. Other resources, such as disk read and write, are left in RAM while the operating system is running because such resources are more often used than others.

The inherent problem with the prior art architecture depicted in Fig. 1 is that resources, such as RAM, or disk, are shared by programs simultaneously, giving a rogue program a pipeline to access and corrupt other programs, or the O/S itself through the shared resource. Furthermore, as the applications that are to be used in the prior art are of general nature, many features are made enabled to them by the O/S, thus in many cases bypassing the O/S security mechanism. Such is the case when a device driver or

daemon is run by the O/S in kernel mode, which enables it unrestricted access to all the resources. Corruption can thus occur system wide.

## SUMMARY OF THE INVENTION

Accordingly, it is an object of this invention to solve the problems with existing systems described above.

It is another object of this invention to provide a system and method for isolating multiple computer programs and operating systems executing simultaneously.

It is another object of this invention to provide true protection of multiple computer programs and operating systems executing simultaneously from untrusted and potentially rogue programs and operating systems.

It is another object of this invention to provide a simple, secured sharing environment for multiple computer programs and operating systems executing simultaneously.

It is another object of this invention to provide limited, controlled sharing of data and resources between multiple computer programs and operating systems executing simultaneously while protecting the multiple processes from each other.

It is another object of this invention to prevent defects present in one program or operating system from causing defects in another program or operating system on the same computer.

These objects and other advantages are provided by a method and system for protecting an operating environment on a processor from a first program operating on the processor. The method includes the steps of allocating memory space

for use only by the first program while the first program is executing, allowing communication between the first program and the operating environment through only a single link employing a single method selected from the group consisting of a shared memory space, a dedicated interrupt, and a dedicated I/O port, and managing a restricted operating environment for the first program on the processor, the restricted operating environment preventing the first program from accessing resources on the processor except for the allocated memory space and the single communication link.

In order to create a truly secured sharing environment, the system of the present invention provides a simple, shared environment which allows very restricted resource access. The system limits sharing capabilities to those provided directly by the hardware as opposed to the sharing capabilities supplied by the O/S or other programs, and does nothing except activating and de-activating these sharing capabilities. This results in a truly simple and secured way of running several programs on the same computer. Many aspects of the system may be formally verified, including special I/O routines which implement the format and protocol for data passing between the restricted operating environment and the remaining processing environment as well as the protocol itself.

The system does not allow traditional sharing of resources, such as disk read and write, printer output, interrupts, I/O port access, etc. Although almost any type of computer program may be implemented using the system, it is very effective for programs or operating systems which require high security and limited resources. For example, a computer running several operating systems on the same computer can divide hardware resources between those operating systems using the system of the

present invention. Each operating system would only be allowed direct access to hardware resources which are different than the co-executing operating systems at the same time, while possibly using the restricted link between the programs to share access to the resource in a very controlled and restricted way. With respect to each hardware resource, the trust level or security level afforded each running operating system is different.

This same scheme may also be equally applied to multiple computer programs within one or several operating systems executing on the same computer. Each application is able to receive input, process it and then output results without any other system resource being involved.

This undisturbed hardware resource acquisition allows an implementation of a security policy wherein a first operating system or program has a different trust level or security level than a second or plurality of operating systems or programs which share the same hardware. While maintaining several programs with potentially different trust levels on the same processor, and keeping those programs separated by means of hardware mechanisms provided by the processor, the present invention allows a very restricted, highly controlled means for communication between the programs, again by exploiting mechanisms natural to the processor, thus keeping the communication mechanism relatively simple.

In the discussion below, the term "program" can be interchanged with operating system to describe an alternative embodiment unless otherwise stated for a specific feature. In the first embodiment, the system of the present invention is able to isolate a program executing in a single operating system from other programs executing

in that single operating system. In an alternative embodiment, the system of the present invention is able to isolate an operating system from other operating systems executing on a computer system which may have one or several computer programs executing within each operating system.

## BRIEF DESCRIPTION OF THE DRAWINGS

For a fuller understanding of the invention, reference is made to the following description taken in connection with the accompanying drawings, in which:

Fig. 1 is a ring diagram representing system layers of a prior art O/S kernel and application program;

Fig. 2 is a diagram representing system layers of a first embodiment of the present invention;

Fig. 3 is a schematic overview of the operating environment of the first embodiment of the present invention;

Fig. 4 is a flow diagram representing major operations for implementing the first embodiment of the present invention;

Fig. 5 is a detailed flow diagram representing system control of the first embodiment of the present invention;

Fig. 6 is a ring diagram representing system layers of a second embodiment of the present invention;

Fig. 7 is a flow diagram representing major operations of the second embodiment of the present invention;

Fig. 8 is a detailed flow diagram representing system control of a second embodiment of the present invention;

Fig. 9 is a block diagram of a gateway system employing the security architecture of the present invention;

Fig. 10 is a block diagram of a gateway system connected between an internal and external computing environment of one preferred embodiment of the present invention;

Fig. 10b is a block diagram showing an alternative architecture for a gateway system connected between an internal computing environment connected to the Internet in accordance with the present invention;

Figs. 11a and 11b are block diagrams of the external and internal robots, respectively, shown in Figs. 10a and 10b;

Fig. 12a is a flow chart showing a process of processing incoming data performed by the apparatus of Fig. 10a in accordance with an embodiments of the present invention;

Fig. 12b is a flow chart showing the process shown in Fig. 12a in greater detail, referring to elements of the block diagrams shown in Figs. 11a and 11b;

Fig. 13a is a flow chart showing a process of processing outgoing data performed by the apparatus of Fig. 10a in accordance with an embodiment of the present invention;

Fig. 13b is a flow chart showing the process shown in Fig. 13a in greater detail, referring to elements of the block diagrams shown in Figs. 11a and 11b;

Fig. 14 is a flow diagram of the Protocol Manager module shown in Figs. 11a and 11b;

Fig. 15 is a block diagram of the object repository and a session sub-module for the apparatus shown in Fig. 14;

Fig. 16 is a flow diagram showing a process for converting data from an application protocol to a simplified internal protocol in accordance with one embodiment of the present invention;

Fig. 17 is a flow diagram showing a process for converting data from a simplified internal protocol to an application protocol in accordance with one embodiment of the present invention; and

Fig. 18 is a sample of a protocol entity table shown in the apparatus of Fig. 14.

In these drawings, like items are assigned like reference numerals.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

With reference to Fig. 2, in a computer system 300, a first embodiment of the present invention comprises two entities, a master software program 202 controllably acting as a protective container, and a restricted operating environment or cell 204 acting as a RAM segment container through which master 202 monitors and controls programs executing inside the cell 204 container. Master 202 protects other processes, such as the computer's O/S 200, from a program executing inside cell 204. The system may be implemented on a Windows/NT system in VM86 switchable mode. A system implemented using Windows/NT may run on Intel's CPUs, 80386 and above. Master 202 switchably operates along side O/S 200 at the same priority and system level as O/S 200.

A process master 202 constructs a cell 204 and loads a program into the cell 204. Master 202 is responsible for executing the program inside cell 204, and passing data from cell 204 into an outside environment 302 comprising O/S 200, master 202, and other programs and resources such as peripherals. Master 202 is also responsible for shutting down the program and dismantling cell 204.

With reference to Fig. 3, a schematic overview of a computer 300 is depicted with an exemplary system of the present invention installed and running. Computer 300 includes an outside system environment, generally indicated at 302 which includes an operating system 200.

The processor in the following examples is assumed to be an Intel Pentium, although the examples are basically applicable to Intel's model 80386 and 80486 processors, and with the necessary modifications known to those of skill in the art, to various other CPUs.

The system depicted in Fig. 3 is executing a program in cell 204 which communicates with outside environment 302 using a link, or vector, 304, which comprises, in this example, shared memory. The program executing in this example is a TCP/IP stack driver for communicating with a hardware component, such as a network communications card. The system of the present invention is excellent for execution of such hardware driver programs because of the complexity in their software which may have many bugs and potential software conflicts with programs in outside environment 302. Master 202 acts as a mediator between an executing program in cell 204 and the outside environment 302. The exemplary program 306 executing in cell 204 includes TCP/IP communications stack which queues data for routing to an outside

resource, which in this case is a network communications card. Master 202 allows passage of the information only through the well-defined, narrow, controllable link or vector 304, described in detail below.

Outside environment 302 may comprise hardware peripherals, operating systems or other computer programs. The environment is protected from the program executing in cell 204 by master 202. However, the program is able to send and receive information from environment 302 through vector 304.

Master 202, is a process in outside environment 302, executing as a privileged task, such as a device driver, and has potentially full control over at least one CPU. Master 202 may isolate cell 204 from outside environment 302, so that outside environment 302 is not aware of cell 204. O/S 200 grants master 202 an execution time-slice, during which master 202 may access resources such as the an interrupt description table (IDT) 308, which maps each interrupt to its handler and page directory tables provided in task descriptors set during initialization of the master. Master 202 then grants the time-slice to the program in cell 204. Master 202 provides a connection 304 between outside environment 302 and the program executing in cell 204. Master 202 is also responsible for cleanup once the execution of the program ends. Master 202 dismantles cell 204 after the program terminates.

Cell 204 is the loaded executable program's 306 immediate environment. Master 202 provides cell 204 with virtual interrupt handlers, or stubs, task descriptors or other resources needed within cell 204 for executing program 306. Cell 204 allows the executing program 306 to access only RAM address space assigned to it by master 202 via the page tables provided in the task descriptors loaded during

setup. The means for communication for program 306 with outside environment 302 is through vector 304. Vector 304 may be either an interrupt, I/O port or a specially designated shared memory address space in RAM which can be strictly controlled and evaluated by master 202 before allowing access to the data from vector 304, or allowing the program 306 in cell 204 to access data vectored from other programs or operating system 200.

Although not necessary for operation of the present invention, the program 306 loaded for execution in cell 204 comprises software designed or adapted to run, independent of any service other than vector 304 to outside environment 302. The program 306 software cannot invoke system calls except in the case where the program is an O/S of the second embodiment discussed separately below. In the first embodiment, the program 306 cannot issue interrupts or I/O instructions unless they are part of its controlled link vector 304 to outside environment 302. The program 306 is thus confined in cell 204, and serviced by master 202.

However, even with the most CPU intensive programs, certain resources in outside environment 302 communicate with the program 306 executing in cell 204. For example, when the clock interrupt handler 310 of O/S 200 is invoked, master 202 must communicate the event to the program 306 executing in cell 204. Likewise, when a hardware (H/W) interrupt is invoked, the H/W interrupt handler of O/S 200 associated with the particular hardware event must communicate the event to the program 306 executing in cell 204. Master 202 provides a link or vector 304 between the program 306 executing in cell 204 and the outside environment 302 which is simple and restricted in order to keep implementation as simple as possible. The H/W interrupt is

needed for communication with hardware peripherals, such as a network card accessed by the exemplary TCP/IP program 306 of Fig. 3. When a program 306 is executing in cell 204 which does not require communication with H/W peripherals, then there is no need for such H/W handler, which further simplifies operation of the system of the present invention.

One method for providing link 304 is by use of a passive memory link. The passive memory link is the most secure method, but is also the most restricting one. With a passive memory link, master 202 loads program 306 together with a specially created input buffer, into cell 204. Thus, the input buffer is already in the accessible memory area for program 306 when program 306 begins execution. Likewise, an output buffer is extracted by master 202 after program 306 is terminated. The benefit to this method is that there is no interaction, between program 306 and entities of outside environmental 302. The drawbacks to this method is that program 306 is not interactive with outside environment 302 during execution of the program 306. Communication is thus provided with only two transactions, an input to program 306 at the beginning of execution, and an output at termination of execution.

An alternative method for providing a link vector 304 between the program 306 and outside environment 302 is through a small shared memory segment with or without interrupts. A predefined part of the memory assigned to the program 306 to serve as shared memory. The area is used for communication between the program 306 and outside environment 302, or with master 202 itself. The program 306 may write data onto the shared memory, indicating the completion of the write by either flagging a bit in a preset position in a shared memory control block, or by invoking an

interrupt as discussed below. Outside environment 302, or master 202, may later read the data, flagging its completion in a similar manner. Reading data by the program 306 is accomplished in similar fashion.

The shared memory link method may be implemented using dedicated interrupts. As part of cell 204, the master may install special link interrupt handlers, or stubs, for cell 204. The I/O stubs are responsible for moving data from the program's 306 allocated memory space to memory space in the outside environment, or master 204, and vice versa. The program 306 can perform I/O by invoking standard interrupt calls.

Another alternative method for providing a data link between the program 306 in cell 204 and outside environment 302 is through use of dedicated I/O ports. As part of cell 204, I/O ports can be reserved for the program 306. The I/O ports may be connected to a peripheral device such as a serial bus, parallel bus, NIC or SCSI device, etc., which in turn is connected to another peripheral device, connected to an I/O port for outside environment 302. Alternatively, to gain more security, the I/O port for outside environment 302 may be connected to master 202. In this way, the program 306 communicates with outside environment 302 by reading data from I/O port(s) and writing data to I/O port(s).

By using any of the above link methods, outside environment 302 may send/receive data either directly, or through master 202. Operating through master 202 has the advantage of being more secure, as master 202 becomes a mediator to the data arriving from the program 306. It is therefore possible, especially with respect to the second embodiment to have two programs effectively using a single resource, although

in a very restricted, controlled manner, through a link as described above, and not transparently. This mechanism may be implemented in the by assigning a resource to a first program, which has complete control over it. This program may relay data to and from the resource to a second program using the restricted link between the two programs. Both programs are aware of this mechanism and take the necessary steps in its establishment and maintenance, making the mechanism non-transparent.

With reference to Fig. 4 the flow diagram depicted represents the major steps to implementing the first embodiment of the present invention. The first step is to initialize and load master 202 into memory which gains complete control over the CPU and sets up needed data, step 400. Next, each program 306 cell 204 is initialized, step 402.

Step 402 comprises several sub-steps. For each cell 204, an executable program 306 is loaded into physical memory. Memory is allocated for each cell 204, as well as hardware resources, CPU timing, interrupt vectors and any other resource supported directly by the CPU for the program cell 204. Master 202 then constructs each Cell by first setting up the CPU's protection, including I/O permission tables, virtual memory tables, interrupt re-direction maps, task tables, segmentation tables, etc., to isolate the program 306 from any memory or hardware except that allocated to it by master 202. The CPU's scheduling mechanism, or clock interrupt, is then set up to distribute CPU time between cells 204 and outside environment 302. The scheduling mechanism is then started.

Execution of each program 306 is then begun within each cell 204 as the CPU's scheduling mechanism allocates a time slice to each cell 204, step 404. As each

program 306 executes, various requests from the program 306, e.g. I/O, are serviced, step 406. As services are provided, outside environment 302 is protected by disallowing each program 306 to access resources other than its allocated memory and the preset resources assigned to it, step 408. In case of an attempt to violate these restrictions, master 202 can terminate the program 306, or ignore the attempt, or take any other appropriate measure programmed into master 302. Hardware interrupts are handled by master 202, step 410, as explained in detail below, as well as clock interrupts, step 412. When a program 306 terminates, master 202 shuts down cell 204 and resets the memory allocated to cell 204, step 414.

In order to make it easier to confine a program 306 in a cell to only CPU level services a program 306 may be especially adopted for executing in cell 204. Software written from scratch can be coded to take into account the limited sharing environment of the present invention. For existing programs, the source code may be modified to eliminate all calls to services at the level of O/S 200. Otherwise, if the program 306 running in cell 204 attempts to use resources in outside environment 302 not assigned to it, master 202 may invoke an interrupt stub to ignore the request. For example, a program 306 that calculates an output buffer from an input buffer may attempt to write messages to a screen while it's processing the data when the screen has not been allocated to cell 204 by master 202. In such a case, where the attempt to violate the restrictions is expected, an interrupt stub for the resource, in this case, the stub for the write-to-screen interrupt, can be constructed such that instead of terminating the program 306 and/or informing outside environment 204, the interrupt stub simply ignores the request, emulating a successful operation, and returning control

to the program 306. In this example, the interrupt stub for the write-to-screen interrupts should return to the program 306 with a successful status, emulating a successful write-to-screen, where in fact, nothing has happened. In this way, the program 306 may finish its task, terminating with the desired results, possibly in an output buffer. In this way, a program 306 can be executed in a cell without modification of the program 306.

With reference to Fig. 5, a control and data flow is depicted for the first preferred embodiment of the present invention. After the Windows/NT system is up and running, the system performs an initialization of a master 202, step 500. The system allocates memory for master 202 and a cell 204, step 502. The master creates a Windows/NT standard VM86 task, step 504. Next, master 202 loads a program 306 for execution into the cell 204, step 506. Master 202 then may define an I/O vector or buffer 304 for the program 306 to have limited contact with outside environment 302, or master 202 itself, step 508. The I/O buffer 304 may either comprise an interrupt, I/O port or a specially designated shared memory address space in RAM which can be strictly controlled and evaluated by master 202 before allowing access to the data vectored by the program 306 to other programs.

The last step in initialization comprises replacing the normal computer interrupt routines for the program 306 in cell 204 with interrupt handlers or stubs, step 510. The interrupt stubs can handle interrupt calls performed by the program 306 in a number of different ways which keeps the program 306 isolated from outside environment 302. For certain kinds of interrupts, such as calls to write to areas of the computer's physical memory addresses, an interrupt stub for that interrupt may terminate program 306 execution for cell 204 and return an error in any output buffers

for the program 306. For other kinds of interrupts, such as screen output which may not be as crucial to the program 306 execution itself, an interrupt stub may handle the write-to-screen interrupt call by simply ignoring it. Different cells 204 may require different interrupt stubs depending on the program's 306 purpose and level of security with respect to the outside environment 302.

The memory allocation for a particular cell 204 may vary. With today's applications, one megabyte (1M) or more of memory may need to be allocated. Other programs, such as standard MS-DOS programs, require less than 1MB to execute. Step 502, comprising allocating memory also comprises creating a virtual memory mapping table. When the program 306 loads, the program 306 is not aware of any memory above or below the space allocated in RAM for the particular cell 204 the program 306 has been loaded in. To the program 306, the lower end of memory is, for example 0, and the upper end of memory equals the amount of space which has been allocated to cell 204. The table translates memory access interrupt calls from the program 306 to the physical memory address so that only the data allocated to that cell 204 is read or written.

To complete step 504 the master allocates a VM86 CPU task by inserting a task state segment (TSS) entry into a global description table (GDT), which contains pointers to TSS entries and shared memory segments or other loaded resources. A register within the TSS which points to the physical memory location of the page directory for the task for cell 204 is set. The VM86 bit in the EFLAGS register of the TSS is set so that the task is run in VM86 mode, as well as the current privilege level (CPL) for the TSS, which is set to 3, the least privileged level. A CPL

of 3 means that the TSS is unable to modify its cell. The privilege level for the task or interrupt gate is similarly set to 3 for master services.

After the program 306 is loaded into cell 204 and ready for execution, the master requests a time-slice from Windows/NT for the cell 204, step 512. Once the time-slice is granted, control moves to the program 306, step 515, within cell 204 for its CPU time share, step 514. The program 306 executes as a VM86 task within cell 204, using standard long-jump instructions.

As with programs running in outside environment 302, a program 306 within cell 204 may trigger several events, or several events may interrupt execution of the program 306. One of those events is that the program 306 may encounter or produce a software I/O interrupt, step 516. If an I/O interrupt occurs, control is passed to the corresponding interrupt stub for an I/O installed in step 510. If cell 204 is one which allows limited input and output through an I/O interrupt, the cell stub may relay the request through Windows/NT to perform the I/O operation, step 520. Before doing so, a routine may be in place which checks the request to make sure it will not cause damage to Windows/NT or any other running applications in outside environment 302.

After processing the request, master 202 sends control back to the I/O interrupt stub for cell 204, step 522. If the I/O operation was a request for data, then master 202 routes the data from shared memory 304 to the program 306, step 524. The program 306 receives the input, if requested, and continues execution, step 514.

A hardware (H/W) interrupt may be produced while the program 306 is executing, step 526. Control is passed to the interrupt stub for the particular H/W interrupt produced, step 528. If the H/W interrupt is one which is allowed during

execution of the program 306, cell 204 passes control to master 202, which then passes control to the particular Windows/NT routine, step 529, which services the H/W interrupt by switching the CPU back to NT mode, step 530. After the H/W interrupt is handled, control is passed back to the interrupt stub in VM86 mode, step 532, which in turn passes control back to the program 306, step 534.

The time slice allocated to the VM86 task may expire producing a clock interrupt, step 636. The interrupt stub for a clock interrupt receives the interrupt call, step 538. The clock interrupt is serviced by returning the CPU to NT mode and passing control to Windows/NT, step 540.

The program 306 may naturally terminate, step 542. The master then cleans up and returns allocated memory and resources used by the cell 204, and then releases control back to NT, step 544.

A program 306 may try to perform a prohibited operation, step 542. If this occurs, the CPU issues a general protection fault (GPF), step 546. Control is kept within master 202 and handled by shutting down the program 306 so as not to disturb other programs which are running under in Windows/NT, step 548. The master terminates execution of the program 306 and reports back to NT, step 570. Alternatively, the system may ignore the violation and return control to the program 306 for execution of further instructions.

With reference to Fig. 6, a second preferred embodiment of the present invention is employed to isolate several operating systems executing on one physical computer. An O/S monitor 600 gains full control of the computer. A monitor 600 can

selectively enable or disable access to I/O ports and other memory locations. For example, monitor 600 can disable access to the I/O port for a floppy disk controller.

The second embodiment comprises an O/S layer 602 above monitor 600 which contains one or more simultaneously executing operating systems. User programs may execute in a an application layer 604 above the O/S layer 602.

An operating system is a particular type of program that does not assume O/S services. In the second embodiment of the invention the system uses monitor 600 as a mechanism which allows an O/S to assume it has full control over the computer by creating a virtual environment for the O/S.

The system traps any attempt of the confined O/S to re-program the security feature of the CPU, or any other O/S level feature, such as virtual memory. Monitor 600 services these attempts in a way similar to master 202 of the first embodiment described above. The system further traps any attempt to receive information about the state of the CPU's security features, and the monitor emulates the results of the CPU instruction which was supposed to be executed, without actually modifying the security configuration. The master may have to replace subroutines for an O/S because some CPU instructions on certain systems return information about the CPU state which cannot otherwise, without those replacements, be trapped. For example, an O/S such as Windows/NT running on Intel-Pentium may query the CPU to find the current privilege level in which it is running, expecting the result "0", which is the most privileged ring of execution. As this query is not trappable, each such instruction within the O/S is replaced with a call to a specially designed routine which emulates the desired result of 0.

With reference to Fig. 7 the flow diagram depicted represents the major steps for implementing the second embodiment of the present invention. At computer system boot-up, the first step is to initialize and load monitor 600 into memory, and execute it so that it gains complete control over the CPU, step 700. In step 700 CPU tasks are allocated by monitor 700 as explained in more detail below. Each O/S is loaded into their allocated task in step 700.

Execution of each O/S is then begun within each task as the CPU's scheduling mechanism allocates a time slice to each O/S, step 702. As each O/S executes, various requests from the O/S, for example I/O, are serviced, step 704. As services are provided, other O/S tasks are protected by disallowing each O/S from accessing resources other than its allocated memory and the preset resources assigned to it, step 706. In case of an attempt to violate these restrictions, monitor 600 can terminate the program, or ignore the attempt, or take any other appropriate measure programmed into monitor 600.

In the system of the second embodiment, an O/S may have read only access to special shared memory segments of other O/Ss, step 708, as explained in more detail below.

A clock interrupt is filtered through the monitor to the O/S running in the current time slice, step 710, or is trapped by the monitor, signaling it to move control to a different O/S.

When an O/S terminates, step 712, the monitor shuts down the task the O/S was executing in and resets the memory allocated to the task.

If the last O/S terminates, step 714, the monitor may shut down or

change to configuration mode.

With reference to Fig. 8, a control and data flow is depicted for the second embodiment of the present invention. Initialization of the system occurs when the computer boots up, step 800. The monitor initially is inserted as the lowest layer above the computer's hardware interface.

The configuration information for each operating system (O/S) is read into the monitor, step 802. A virtual environment for each O/S is initialized which mimics the actual computer system's firmware. Alternatively, if a particular O/S is not compliant with the specific hardware of the computer system, an emulation layer can be installed by the monitor for that particular O/S.

Tasks are initiated with allocated memory pages for each O/S which is to be loaded, step 806. As in the first embodiment, each memory page is installed with a virtual memory address table so that the O/S is not aware of any other memory space other than that allocated to it. In this step, the monitor sets up low level segmentation, interrupts and any other security and control mechanism for the CPU. For each O/S a task is setup with a low permission level, either 1 or 2. The O/S is given an execution privilege which is still more privileged than the user applications running within the O/S itself, which can receive a permission level of 3. However each O/S receives a permission level which gives it less privilege than the monitor itself, which runs at the most privileged execution layer of the system, level 0. Each O/S can be allocated CPU time as with application programs in the first embodiment, step 808. A time scheduler to distribute time between the O/Ss is setup, step 810.

Next, the O/S images are loaded into their respective task spaces, step 812. As in the first embodiment, limited shared memory may be set up between O/Ss, step 814. However, the shared memory is not activated in this step. Further, protection of critical tables, such as GDT and IDT page directory and page tables is provided by the monitor, step 816. If available, this protection can be achieved by using a CPU paging mechanism such as the paging mechanism built into Intel's PENTIUM processor.

Next, exception and interrupt handlers replaced for each O/S cell, step 818. This step is similar to step 510 in the first embodiment described above. However exception handlers for operating systems are more extensive than for applications. For example, a general protection exception handler is installed to enable the Windows/NT O/S to run in layer 602 (Fig. 6) by carefully emulating privileged instructions for the Windows/NT O/S on behalf of the CPU. A page fault exception handler implements the protection on critical tables to trap references to memory areas which do not belong to the O/S causing the page fault.

Once the O/Ss are loaded and ready for execution, the first O/S on the time slice stack will retain a time slice from the CPU, step 820. The monitor's scheduling mechanism grants time slice for an O/S, causing it to run in its own virtual environment. If this is the first time the O/S runs, then the virtual environment emulates a user or automatic boot-up sequence, as expected by the O/S. Otherwise, the O/S resumes running where it was stopped, step 824.

As with the first embodiment, a number of events may occur during a running O/S's CPU time slice, one of which may be a H/W interrupt, step 826. The

interrupt stub for the particular hardware must determine which O/S the interrupt should be routed to, step 828. This may depend on the hardware resource which the specific piece of hardware is allocated to. In the second embodiment, service of the interrupt is assigned to the O/S which the hardware is servicing, step 830. Control is then passed back to the monitor interrupt stub, step 832, which passes control back to the O/S which has been allocated the current time slice, step 834.

Similarly to the first embodiment, a clock interrupt causes the current O/S to stop, step 836. However, unlike the first embodiment, control is transferred to the next O/S, step 838. The monitor simply arbitrates which O/S is to receive the next time slice, step 840.

Any attempt to modify the virtual environment by an O/S, step 842, such as changing an interrupt or modifying a virtual page directory, causes the CPU to generate a general protection fault, step 844. This fault is handled by the monitor, which emulates the modification so that it appears to the O/S that the modification was successful, step 846.

In case a reference or request is made to the virtual environment it can be trapped, or intercepted by a CPU mechanism. There are interrupts that may be activated as a result of such an attempted reference. In a case where there is no way for CPU to trap references to the environment, for example using the segment register to determine the current permission, the O/S must be patched by replacing the problematic instruction with a call or branch to special code which emulates the action, returning the expected result to the O/S so that the monitor can be used to confine it. If the O/S references the segment register directly, it may not be trappable/interruptable. In this

specific scenario, the monitor's existence may not be transparent to the O/S because the segment register contains information which discloses the fact that the O/S is running with a monitor, and is less privileged than expected. So, the O/S may not behave well after receiving this information. One way to resolve this is to completely remove the instruction from the O/S which discloses the information. For example, the read-segment-register instruction may be replaced with a call to a monitor routine which emulates the call so that the routine provides the expected result. This type of patch is made prior to the execution of the code, e.g. at the binary, or just after loading the O/S image, in the memory image of the O/S.

Each O/S has a shared memory page for which only it has write privileges, and can be read by any other O/S. For an O/S requesting data from another O/S to take advantage of this page, it sends a request to the monitor to activate it by invoking a software interrupt, step 848. A special driver is loaded which facilitates information sharing between the O/Ss, step 850. The requesting O/S may initiate a finish interrupt once it receives the data, step 852. Control then passes back to the monitor's data-share interrupt routine, step 854, which finishes execution. The monitor copies the result buffers, if any, to shared memory, step 856.

When the O/S invokes a halt (HLT) instruction, step 870, the monitor ceases to allocate time for the calling O/S, step 872. The monitor ceases servicing the specific O/S invoking the HLT instruction, and may possibly restart it, step 878. If all O/Ss shutdown, step 874, the monitor may either shutdown the computer or, alternatively, move to configuration mode, step 876. This mode can also be activated by a special hot-key combination.

Both the first and second embodiments of the present invention can be implemented on any processor supporting isolation and/or protection mechanisms such as privilege levels, virtual machines and paging memory. Intel's 80386 or higher (80386+) protected mode and Intel's 80386+ VM86 mode is supported by the system. The monitor supports any operating system consistent with protective mode that is used for the isolation. For example Linux, Windows/95, Windows/NT, Solaris/x86, all operate on Intel's 80386+ protected mode. MS-DOS cannot be used in Intel's 80386+ real-mode, because no protection/isolation is offered in this mode, but the system may be implemented using VM86 mode for MS-DOS.

The system of the present invention can be used with other, similar processors. It should be understood that the system of the present invention is not restricted to the Intel 80386+ architecture.

Programs which are CPU intensive work best with the system of the present invention because of the reduced need for environmental resources, although other programs which are less CPU intensive may be executed within a cell. The preference is to limit operating system services needed to for each cell. The system works best with programs' for which, once all the required input is given, most of the programs' activities occur within each cell's allocated memory segment, while using limited I/O functions, if at all. A good example of an application which executes best with the system of the present invention is a protocol processor, such as an SSL decryptor or HTTP parser, because it is CPU intensive, while having low demand for I/O other than an initial input buffer and final output buffer.

With reference to Fig 9, and as described in greater detail below, the system of the present invention may be used to operate two or more programs which form a security gateway system for protecting an internal trusted network from the external environment. An external program or robot 6 receives messages from the external environment 7, converts the content of these messages to a simple, harmless form, and passes them along to an internal robot 3 which converts the simplified form of the content to a form usable by applications in the internal network 2. The external and internal robots 6, 3 may be operated on a single processor while maintaining security of the internal network 2 using the systems described herein. For example, the external robot 6 may be operated in the restricted operating environment so that attacks on the external network will not proliferate into the rest of the operating environment including the internal robot. Alternatively, both the external and internal robots 6, 4 may be operated in restricted environments, thus providing further protection.

The two robots 4, 6 are implemented on a single CPU using a protected mode such as the VM86 mode or the restricted operating environment provided by the master or monitor programs represented as VMM program 9 and Pentium technology. For example, in a single CPU running the Windows NT™ (WINNT) operating system, each robot, or at least the external robot 6, is operated in protected mode under the supervision of a monitor program which prevents each robot from affecting the operation of the other and the rest of the CPU's environment. The monitor program also negotiates the communication of data between them, implementing the communication channel or bus 4 between them using shared memory resources 5 and a special API for each protected mode. Thus, the two software robots 4, 6 are separated

by the CPU under the control of the VMM program in a way that each robot is assigned some resources of the computer 1, 8 (such as disk space, memory address range, peripheral devices like floppy disks or tape drives) which are not shared with the other robot, and the policy of separation is enforced by the VMM program. Only one resource, the communication bus 28, is shared by the two robots 4, 6, and this bus 4 may be implemented, for example, by a dedicated memory address space.

The VMM program 9 and the robots 4, 6 running in their private environments may be executed on a dedicated computer. They may also run on a non-dedicated computer, in which case certain modification to the standard OS (e.g., Windows) might be necessary in order to force it to run in a protected mode. The VMM program 9 controls all the events at the CPU level, and enforces the two virtual processing entities on a single CPU machine by hardware interrupts.

Embodiments of the architecture and operation of the gateway system will now be described in greater detail with reference to Figs. 10a-18.

Referring to Fig. 10a, a network security gateway 10 is connected between an internal computing environment 12 and an external computing environment 16. In the embodiment shown in Fig. 10a, the internal computing environment contains a web server 13, which may belong to a web subnet, and a sensitive system server 14. The external environment 16 may be any environment external to the web and system servers 13, 14, but typically includes the Internet. The gateway 10 is connected to the internal system server 14 via communication bus 20a, to the web server 13 via bus 20b, and to the external environment via bus 22. These buses may be implemented as Ethernet connections, using conventional network interface cards such as Ethernet PCI

cards, or may be implemented as serial connections using a V.35 interface. Other connection methods may be used as known to those of skill in the art. The buses 20a, 20b, and 22 may use the same or different types of connections.

The security gateway 10 contains two separate and distinct processing entities 24, 26, referred to herein as robots, connected via a dedicated, secure communication bus 28. The internal robot 24 is connected to the web server 13 and system server 14 via the buses 20b, 20a, respectively, and the external robot 26 is connected to the Internet or other external environment 16 via the bus 22. As described in greater detail below, each robot is capable of translating or reducing a communication or message received from the respective environment to a simplified message using a simplified protocol format referred to herein as a clear inter-protocol or CIP, transmitting the CIP message to the other robot using the inter-robot bus 28 using an inter-robot transfer protocol or IRP, and translating such CIP messages received from the other robot into messages formatted for the respective environment.

In one embodiment, the security gateway 10 is connected to an organization's internal networks in the following manner. An application proxy is connected through bus 20a to the internal system server 14, and a web proxy is connected through bus 20b to the web subnet wherein the web-server 13 resides. As one skilled in the art will appreciate, it is possible to have multiple web-subnets, with a dedicated interface per each, or to have several web servers in the same subnet; likewise it is possible for the apparatus to serve multiple internal environments with a dedicated interface per each, and also several servers and/or applications in the same internal zone. The embodiment described with reference to Fig. 1a services a single

internal environment with a single server running a single application, and a single web-server in a single web-subnet, for reasons of simplicity and ease of implementation. However, the principle of replication and extension of the gateway system 10 is a system design parameter understood by those of skill in the art.

For example, Fig. 10b illustrates an alternative architecture which may be used for the internal environment 12a. As shown in Fig. 10b, a LAN server 13a is connected to the internal robot 24 via several interfaces 20a, 20b, 20c. The LAN server 13a services communications for a number of internal application servers 14a-14f, including for example an SQL database server 14a and a banking server 14b having its own additional server security process 11 that provides access control and other security measures. The three interfaces 20a, 20b, and 20c provide for a variety of communication protocols to be used, including one 20a for issuing SQL commands to the SQL database server 14a, one 20b for transmitting email and web communication protocols including CGI calls within HTML data, and one 20c for high security financial communication protocols specific to the banking server 14b. Corresponding multiple interfaces 22a-22c may be provided between the external robot 26 and external environment 16 to receive message having various communication protocols.

There may be a separate processing module for each message protocol or this combination may be streamlined to provide one module for the "gateway" transfer protocols implemented in the security gateway, one for "middleware" protocols that bypass the web server, perhaps also one for encryption protocols and one for applications protocols. The streamlined alternative does away with the system overhead incurred by the double-filtering of HTTP-protocol messages that otherwise

occurs to less than that of a linear addition, while being free of obvious security flaws or leaks and still blocking tapping. On the other hand, a very "strong" machine with advanced operating system performance and features, such as a SUN SPARC station is required to run it, adding to its overall cost for low-end users. The advantage of the modular design is the economies of being able to selectively implement additional protocols by adding individual modules specific to the tasks at hand.

The internal communications bus 28 connects the respective robots 24, 26 in accordance with the serial bus, parallel bus or universal serial bus standard. In accordance with the present invention the internal bus 28 linking the two robots may, alternatively, be a SCSI bus, fiber-optic, a network interface link or even a radio link, where the robots must operate over a greater distance, VMM-protected shared memory, or the like.

Together, these three elements 24, 26, 28 implement the protection provided by the network security gateway 10 for the protected internal environment 12. The robots 24, 26 are two separate and independent logical processes that execute routines defined by respective security gateway software packages. The robots 24, 26 may be installed on two separate processing devices or one a single processing device operating the one or both of the robots 24, 26 in protected mode.

In some embodiments, the respective software packages are installed on two or more respective separate CPUs, for the sake of simplicity and off-the-shelf component availability. In this approach, each robot runs on a single independent computer processor with non-shared resources assigned to it (for example, disk space, memory address, network adapter, various peripherals, and the like). The only shared

resource in this approach is the communication bus 28. Several configurations may be used to implement this approach. One such configuration is different independent computers (PCs) connected by a communication bus like a serial line, SCSI line and the like, with each PC having at least another network adapter for communicating with the internal network or the rest of the external world. Alternatively, one robot program may run on a computer (PC) and the other on an add-on card, which may be a dedicated card or device, or a standard card (like Intel 80x86 add-on card), installed in one slot of the PC, with this slot serving as the communication bus 28. Both the PC and the add-on card have at least one additional network adapter for communicating with the internal network or the rest of the external environment. The two robot programs may also be run on different independent processors implemented on a standard (e.g., dual Intel 80x86 processors card) or a dedicated add-on card. These two processors are connected by a communication bus like a SCSI line, IDE bus, PCI bus, and the like. Each robot also includes at least another network adapter for communicating with the internal environment 12 or the external environment 16. This add-on card is installed in a standard or dedicated network communications device like a router, bridge, communication server, and the like.

In other embodiments, the two robots 24, 26 are implemented on a single CPU using a protected mode such as the VM86 mode provided by VMM and Pentium technology. For example, in a single CPU running the Windows NT™ (WINNT) operating system, each robot, or at least the external robot 26, is operated in protected mode under the supervision of a monitor program or "mediator" which prevents each robot from affecting the operation of the other and the rest of the CPU's environment.

The monitor program also negotiates the communication of data between them, implementing the communication channel 28 between them using shared memory resources and a special API for each protected mode. Thus, the two software robots 24, 26 are separated by the CPU under the control of the VMM program in a way that each robot is assigned some resources of the computer (such as disk space, memory address range, peripheral devices like floppy disks or tape drives) which are not shared with the other robot, and the policy of separation is enforced by the VMM program. Only one resource, the communication bus 28, is shared by the two robots 24, 26, and this bus 28 may be implemented, for example, by a dedicated memory address space.

The VMM and the robots running in their private environments may be executed on a dedicated computer. They may also run on a non-dedicated computer, in which case certain modification to the standard OS (e.g., Windows) might be necessary in order to force it to run in a protected mode. The VMM program controls all the events at the CPU level, and enforces the two virtual processing entities on a single CPU machine by hardware interrupts.

Structure and Operation of the External and Internal Robots

The external and internal robots 26, 24 are now described in more detail with reference to Figs. 11a and 11b.

Referring to Fig. 11a, the external robot 26 contains a channel manager 4a for wrapping outgoing CIP messages to the internal robot 24 in the inter-robot protocol or IRP and removing the IRP from incoming messages. The external robot also contains a network proxy 4e which wraps messages in TCP/IP or other transfer protocols used in the external environment 16. These protocols may include TCP/IP,

UDP, SPX/IPX, HTTP, SNA, NCP, CORBA, RMI, RPC, or communications transfer protocols. The CIP is also specific to the application protocol used, such as SMTP, POP3, SQL, CGI, and applications-specific protocols such as those used in banking. Because the protected environment 12 may use the secure hypertext translation protocol (S-HTTP) over TCP/IP (or likewise security scheme, e.g. SSL) and structured query language (SQL) over TCP/IP, as well as a specialized financial communication protocol and application protocol, the robots may need respective complementary sets of at least three CIP protocol stacks.

The external robot further contains a routing manager 4b for routing CIP or application format messages between the various elements of the external robot 26, a protocol manager 4c connected to the routing manager 4b for reducing a message from the application format received from the external environment 16 to the CIP in accordance with procedures described further below, and a Communication Layer Security (CLS) routine 4d which provides decryption and authentication services for the security gateway 10 under the direction of the routing manager 4b. The routing manager 4b first forwards application messages to the CLS routine 4d, and then forwards the authenticated, decrypted data from the message to the protocol manager 4c. The protocol manager 4c reduces the native-application protocols it receives from the untrusted environment into a respective CIP format for its particular native protocol(s). The channel manager 4a in the external robot 26 then moves the CIP formatted data onto the inter-robot communication bus 28.

As seen in Fig. 2b, the internal robot 24 has an architecture similar to the external robot 26. The internal robot 24 thus contains a channel manager 2a similar to

channel manager 4a of the external robot 26, a routing manager 2b, a protocol manager 2c, and a number of proxies depending upon the architecture of the internal environment. Given the internal environment illustrated in Fig. 1a, the proxies include an application proxy 2e and a web proxy 2f.

Data received by the channel manager 2a of the internal robot 24 is forwarded to its protocol manager 2c under the direction of its routing manager 2b to be retranslated from the CIP format back into respective native application protocols, and then the retranslated result is sent to the internal environment 12 through the application proxy 2e and the bus 20a or through the web proxy 2f to the web-server 13 which responds in a data stream which is sent back to the gateway 10 through the bus 20b to the web-proxy 2f and then re-directed into the protected system server 14.

Although these various tasks are described as being carried out in separate modules in this embodiment, so long as analogous functions are provided so that they cooperate in producing the described result, these tasks may be grouped, divided between different modules or appended to each other or elsewhere. For example, the security functions (as performed for example, by CLS module 4d) can be divided between the internal and the external robot, perhaps in order to provide better security for the cryptographic variables stored in it (keys and signatures).

The communication between the external robot and the internal robot is carried out solely through a dedicated simple inter-robot protocol or IRP, over the dedicated inter-robot bus 28. The data is translated using a security protocol specific to the applications protocol of the data received and internal to the security gateway operation. Applications within the trusted environment can configure the robots to

authorize data flows using selected communications transfer protocols (CTP), such as the simple mail transfer protocol (SMTP), file transfer protocol (FTP) and secure electronic transfer (SET) protocol. Preferably, the CIP assigned to a communication is specific to the CTP in use.

Any breach of the permitted flow sequences by disorderly operating system calls or looping will be trapped and logged. For example, in the file transfer protocol (FTP) a GET command cannot be recognized unless preceded by a successful login sequence including the USER command, followed by a PASS command. Violation of the required flow order will cause an alarm to be logged and terminate the FTP session. The gateway 10 further enforces data flow requirements, since each translator and interpreter pair is a pair of ad hoc transforms derivative of the protocol used in the incoming message and the types of data flow permitted by the security administrator. For example, if an external SMTP user issues the "MAIL FROM" command, the external server will send the ART equivalent of a "MAIL FROM" command, only when it follows a "HELO" command.

Figs. 12a and 12b show the data flow implemented by the apparatus shown in Figs. 10a, 11a and 11b. In overview as shown in Fig. 12a, at the start 40 of the process, communications packets received from the external environment 16 are time stamped and logged by the external robot 26, step 50, followed by the data-security processing functions such as decryption to plain data. Logging is initiated by a synchronous API module within the security gateway on a "write once" media (e.g. CDW). The logging process performs sparse notations of program state changes,

time-stamped message IDs, system errors, access and flow violation attempts, rule firing for each packet, etc.

Generally error and debug entries are kept in greater detail than the message and state entries. Each service module is automatically periodically polled to maintain a complete audit trail of every administrator action, user login/logout, database error, simplified network management protocol trap or alarm. A database of known intrusion patterns is provided and habitual usage patterns of groups within the trusted environment are monitored and the administrator notified of incidents that diverge from that pattern.

For the sake of security, the log is accessed locally only through the internal, trusted terminal. However, external logs are securely copied to the internal record using the internal CIP protocol corresponding to the external log's native protocol and interpreted into an item format distinct from that used by internal entries, to further frustrate counterfeiting. Alternatively, the external logs may be written to a separate system to decrease the overhead imposed on the internal robot by the logging process. A "dual Channel Manager structure" wherein there is an additional Channel Manager in each robot, dedicated for logging messages, may also be used. The software to handle the logging may use the ACE package or any other commercial product for the implementation. The internal and external logs are recorded asynchronously, using a logger daemon, so that logged items go immediately to the written record without waiting in vulnerable queues during thread lockouts or I/O busy states. This can be implemented by an asynchronous wrapper using generic OS logging, such as UNIX's Syslogd & MSEventLogger for errors and violations, and an

ODBC-standard file structure for the transactional information concerning program state and messages. The open data base convention (ODBC) logger has some asynchronous behavior options, but they are not directly applicable, so it may not represent a realistic alternative.

Then in step 60, the plain data is edited to reduce it to clear data and then translated into CIP format, after which the CIP format is sent over the security gateway's internal communication bus 28 to the internal robot 24. At step 70, the internal robot 24 retranslates, and perhaps also reconstructs, the data from its CIP format back to the format native to the application it is addressed to, perhaps introducing some further editorial changes.

In step 90, if the data belongs to a web-session (HTTP) it is first sent to the web-server 12, the web-server 12 may then initiate an application request, sending a response back to the Internet source of the data the web server received, back through that network-security gateway 10, while the data proceeds to its destination, as noted in step 1100. If, on the other hand, the data does not belong to a web-session, such as data communicated directly to the application server, then the internal robot simply sends that data to the application proxy 2e and over the bus 20a to the internal environment 14, comprising step 1100 and finishing the security gateway's security-assurance process at step 1104.

More specifically, referring to Fig. 12b, if a TCP/IP packet, or some other basic unit of data associated with a suitable communications protocol for the media available, is received by a security gateway proxy 4e corresponding to that protocol, through the external bus 22, it is logged in by the external robot at step 51.

This records the packet's ID number and operational state codes representing the transfer steps completed for the packet. However, since this robot is barricaded, detailed logs are not kept.

At step 52 the security gateway proxy 4e, after having removed the encapsulation provided by TCP/IP or by some other transport protocol used for communicating the data to the security gateway, sends the data to the routing manager 4b which forwards it to the Communication Layer Security (CLS) module at step 53. At step 54 the CLS module decrypts the SSL format , if such encryption is present (or any other security scheme, e.g. S-HTTP), of the message and interfaces with and mediates information required by "mechanisms" that authenticate the identity of the sender, if such authentication is needed, thus providing plain application-format data to the routing manager 4b. Preferably, the public key infrastructure is used for the decryption.

The routing manager 4b then sends the plain application-format data to the protocol manager 4c which edits the application data into clear data and translates it into CIP format at step 61. The protocol manager 4c then moves the CIP data back to the routing manager 4b at step 62. The routing manager 4b sends the CIP data on to the channel manager 4a at step 63, which encapsulates the CIP data using the IRP, transmitting this IRP-encapsulated CIP-format data to the internal robot 24 over the internal communication bus 28 in step 64. The IRP transport protocol may encapsulate CIP data originating from different native application and transport protocols.

The CIP-format data, encapsulated in accordance with the internal IRP transport protocol, is received from the internal-communication bus 28 by the channel

manager 2a in step 71. The channel manager 2a removes the IRP encapsulation and sends the CIP-format data to the routing manager 2b in step 72, which sends the CIP-format data to the protocol manager 2c in step 73. It is the protocol manager 2c in the internal robot 24 that finally re-translates the CIP-format data back into its native application format, possibly modifying the data the data in so doing. Thus the plain application-format data decrypted by the external robot 26 from communications received through the external bus, may not be identical to the clear data in native application format that is supplied to the trusted environment 14 by the network-security gateway 10 over the internal communication bus 20a.

Some additional acceptance test, such as tests directed to authorizing particular actions by particular users for access-control, may be applied to the data at this point in order to further verify the legitimacy of the data. Providence is made in the architecture of the present invention to allow for third party integration of processing modules, to enhance to adaptivity and flexibility of the system. These modules, hereinafter referred to as "Plug-In's," are callable at various places in the process flow of the apparatus, both in the external and the internal robots. An example of a useful such plug-in is the access-control as described immediately below, applied at the internal robot after the CIP is interpreted, possibly at the Protocol Entity level as described later with reference to Figs. 17-16.

When such an access control plug-in is provided, the protocol manager controls access control through editing its rules. The protocol manager invokes access control by sending it four query parameters: actor, action, resource and attributes. Context-sensitive testing may be used to identify redundant messages that are received

more than three times from the same source. The access-control plug-in must then also have reading access to the transactions log maintained by the security gateway 10 in order to make such context-sensitive determinations. For this reason, intrusion detection may also use the access-control interface to the log. The access control logic (ACL) may be extracted from monitoring network activity or by extracting rules from the responses of the network administrator to packets parsed under the control of the administrator. The concept of automatic and/or guided, semi-automatic recognition of flow, access-rules, access-lists and valid/invalid data is intended to supply, along with the apparatus itself, a utility which will intercept all the traffic to and from the secured servers, analyze this traffic and produce a list of users, their allowed activities, and other relevant parameters (time and date of the action, etc.). It is also intended to provide a utility which will extract access information from the server itself, be it a Windows/NT server (registry, etc.) or a UNIX station (/etc/passwd, etc.). ACL data is imported into the gateway 10, partly in off-time (initialization), and partly on-line (updates). Security standards implemented may include RADIUS and TACACS RAS standards, the TSS mainframe standard, or the modern alternatives: NIS, NT domain.

If it passes all such tests, at step 74 the protocol manager 2c sends the application-format data back to the routing manager 2b which determines its destination. If the data is to be sent directly to the application, it proceeds to step 1101. If the data was addressed to the web-server, e.g., if its application format is HTTP, the routing manager 2b sends the application format data to the web proxy 2f at step 91, which re-encapsulates it as a TCP/IP packet, or whatever other suitable transfer protocol is in the protocol stacks being used by the web-server. At step 92 the web

proxy 2f finally forwards the re-encapsulated data to the bus 20b. The web-server 12 in step 93 processes the data. It is expected that the web server 12 translates the data to some application format before transferring it back to the apparatus.

For example, when a user invokes a CGI script on the web-server 12 using a CGI request encapsulated in HTTP transfer protocol, and the CGI script translates that request into an application format, e.g. SQL or banking, the web-server 12 transmits the application format back (e.g. SQL query, banking command) to the network-security gateway 10, where the web proxy 2f receives it and removes the TCP/IP encapsulation of the application data in step 94, before sending the application data to the routing manager 2b.

In step 1101 the routing manager 2b sends the application-format data to the application proxy 2e. The application proxy 2e re-encapsulates the application data in TCP/IP, or whatever protocol was used for communicating the data to the network-security gateway 10, and sends the data to the application server 14 in the internal environment 12 in step 1102, whereupon the security-assurance processing in accordance with present invention, for that data, ends at step 1104.

The reverse process performed by the gateway 10 of processing and transmitting outgoing data from the internal domain 12 into the external domain 16 is now described with reference to the flow charts in Figs. 13a and 13b and with continued reference to the elements identified in figs. 10a, 11a, and 11b. Referring to Fig. 13a, processing by the gateway 10 of outgoing data begins at step 110. At step 120, the internal robot 24 receives application data from the internal system 12. At step 130, the destination of the data is determined: if it is originated from an indirect session

(i.e., a session that involves a gateway, such as a web-server 13), then the data is relayed to the gateway (web-server), whereas if the data originated from a direct session (user client communicating directly with the internal system server 14), then the processing of the data proceeds directly to step 150.

While at the web-server 13, step 140, the data is translated and restructured by the web-server 13 in order to be presented in a web format (e.g., HTML page over HTTP protocol), and then sent back to the gateway 10, to be further processed. At this stage, execution proceeds at step 150. At step 150, the data is reduced into CIP format, possibly with some alterations, possibly with some filtering pertaining to the nature of the information, e.g., a "Top-Secret" titled article may not be allowed to pass out of the internal zone. Then the data is transmitted over the communication bus 28 into the external robot 26.

The external robot 26 re-composes the CIP format data back into application format, possibly with some changes to the data, step 160. It then proceeds to perform some communication security tasks associated with the data, such as encrypting it and/or affixing it with authentication data, and finally, the secure data is sent to the external zone 5, step 170, which completes the process, step 180.

A more detailed representation of the process in shown in Fig. 13b. Beginning at step 110, application data arrives from the internal system 12 to the application proxy 2e, step 121. At step 122, the application proxy 2e removes the TCP/IP encapsulation (or whichever protocol used to communicate with the internal system's network) and sends the data (which is in application format) to the routing manager 2b. The Routing Manager 2b determines the destination of the data, step 130,

according to its association with a session. If the data belongs to a direct session, that is, a session in which the client communicates directly with the internal system 14, the routing manager proceeds immediately to step 151. If, on the other hand, the data belongs to an indirect session in which the client communicates with a gateway such as a web-server 13, and the latter relays the information to and from the internal system, then the routing manager 2b sends the data to the web proxy 2f, step 141.

The web proxy 2f encapsulates the data in TCP/IP (or whichever protocol is used to communicate with the web-server 13 or any other gateways employed), and sends it to the web-server 13, step 142. The web-server 13 then processes the data, which is typically a reply to a previous query sent from the web-server 13 to the internal system 14 via the apparatus 10, and represents it in a web-format (e.g., typically, an HTML data "page" over HTTP protocol, all encapsulated in TCP/IP), and sends this data to the web proxy 2f, step 143. The web proxy 2f removes the TCP/IP (or any other protocol used for communication with the web-server) encapsulation, step 144, and sends the application data to the routing manager 2b.

In step 151, the routing manager 2b sends the application data to the protocol manager 2c. The protocol manager processes the data, step 152. This process may include performing several tests and/or modifications, in order to further protect the internal system 12 and carry out the security policy exercised in the internal domain. For example, it may refuse to forward documents or pages according to the information they carry, or it may remove or conceal some information or all based on its content. The protocol manager translates the data into CIP, which may be a different coding

scheme than that of the incoming direction. At the end of step 152, the CIP data is sent to the routing manager 2b. The routing manager 2b at step 153 sends the data to the channel manager 2a. At step 154, the channel manager 2a encapsulates the CIP data with the IRP protocol used for the bus communications, and transmits the data over the communication bus 28 to the external robot 26.

In step 161, the data arrives through the communication bus 28 to the external robot 26, where it is handled by the external robot's channel manager 4a, step 162. The channel manager 4a removes the IRP encapsulation and sends the CIP data to the routing manager 4b. The routing manager 4b at step 163 sends the data to the external robot's protocol manager 4c. The protocol manager 4c translates the data from CIP format into application format, step 164, possibly with some alterations to the data. The data is then sent back to the routing manager 4b. In step 171, the Routing Manager 4b sends the application data to the CLS module 4d, which performs several communication security duties, step 172, such as encryption and affixing authentication information to the data, according to the security model employed (e.g. SSL). The CLS module 4d then sends the secure data back to the routing manager 4b. The routing manager 4b finally sends the data to the network proxy, step 173, where the application or secure data is encapsulated with TCP/IP (or whichever protocol used for communication with the client in the external zone 16), and sent to the external zone 16 using the NIC, step 174. The flow of information from the internal zone 12 to the external zone 16 is thus completed, step 180.

Structure and Operation of the Protocol Managers

The core of the robot operation is the protocol manager, denoted 2c and 4c in Figs. 11b and 11a, respectively. The protocol managers provide translation between, the various application formats used by application protocols that are authorized for use by and implemented through respective CIP protocols in the security gateway 10 and the CIP formats used internally by security gateway 10. The protocol managers 2c, 4c may also perform various other tasks, pertaining to the content of the data, such as access-control.

As shown in Fig. 14, the protocol managers2b and 4b have respective input queues 210, 410, and output queues 250, 450, several analogous processing entities between them, and two common objects. The internal input queue 210 holds data coming from a routing manager 2b, 4b that is in native application protocol format, and the external input queue 410 holds data from the routing manager, which is in CIP application format. The internal output queue 250 holds data going to a routing manager 2b or 4b in CIP application format after having been translated from the native application format into CIP, and the external output queue 450 holds data that was translated from CIP and is going to a routing manager 2b, 4b in native application format 50.

The processing objects between the input and output queues of the respective protocol managers 2c, 4c are session managers 220, 420, which provide workload balancing for their respective sets of session handlers 230, 430, each session handler handling a single session object 240, 440 at a time. The session handler 230, 430 determines where incoming data belongs, which "session", and if no such session is

active the handler initiates one. The respective sets of session objects 240, 440 comprise generic session processors.

By combining the data currently being received by the protocol manager 2c, 4c and session records obtained from the object repository 1300 of the protocol manager 2c, 4c each element in the respective sets of session objects 240, 440 processes a respective session, that is, a respective communication stream received by the security gateway as multiple, not necessarily contiguous packets. However, two sessions are usually combined into a single entity, a "twin-session", whenever there are two coupled sessions pertaining to the same circuit of information flow, namely that one session handles incoming data and the other handles outgoing data. The coupling is necessary in order for both the sessions to be synchronized in the state of the server and the context of the whole circuit. Similarly, a mechanism is provided for the internal robot (more accurately, the session objects of the internal robot) to be able to synchronize the session objects of the external robots, where the session objects of the internal robot will act as master and those of the external robot as slave in order to maintain security.

Each session object 240, 440 may also write data back to the object repository 1300. The session objects 240, 440 also consult a protocol entities table (PET) 1310, as described further below, to determine the sequence order prescribed by the applicable protocol for processing data received by the session object in a format prescribed by that protocol. The session objects each write the output of their respective translations and editing processes to a respective one of the output queues 250, 450.

The protocol managers' shared storage entities, the object repository 1300 and PET 310, hold information that is at least more global or less temporary in nature than what is held in the queues 210, 250, 410, 450. The information in the object repository 1300 is either global to the whole security gateway 10, or at least global for-each user or for each session, or session-wide, that is, global to a whole session as opposed to local information used in a single protocol layer or information used in a single packet. For example, a user-name entry in the object repository 1300 is global to all the communications transmitted between the user and the server. The PET 1310, on the other hand, is global in that it is used to enforce the rules by which a particular session object chooses which protocol entity to employ to reduce the data or reconstruct it.

A block diagram of one of the session objects is shown in Fig. 15. A session object 240, 440, employs various protocol entities 1710 for handling the different protocols encountered within the data within a session. The session object 240, 440 consults the PET 1310 in order to determine which protocol entity 1710 to use next. The protocol entities 1710 deposit information to and retrieve information from the object repository 1300. That done, the session object 240, 440 then calls packers/unpackers 720 corresponding to those same protocols reflected in the selection of the protocol entities 1710, in order to streamline the required information deposited in the object repository 1300 into a sequence of bytes to be output by the session object 240, 440.

The flow of data coming in to the security gateway 10 in application format through the protocol manager 2c and 4c is shown in Fig. 16. The data arrives in

its native application format at step 1500 and is read by the protocol manager 2c and 4c from the queue 210 containing data coming from the routing managers 2b, 4b. This application-format data is then transferred to the session manager 220 at step 1510. At step 1520 the session manager 220 locates an available session handler 230, and sends the data buffer to that session hander.

At step 1530, the session handler 230 scans the sessions currently active or "open", to determine which session the data belongs to before sending the data to the corresponding session object 240 for processing. If the data does not belong to one of the open sessions, the session handler 230 initiates a new session object 240 and sends the data, all this comprising step 1530. The session object 240 begins by storing the data buffer in the object repository (OR) 1300, step 1540. The session object 240 then consults the PET 1310 to get the identity of the next protocol entity 1710 that should be used to process the data, reducing it to clear data in CIP format at step 1550. If other protocol entities are needed to process the data, then the data is handed on to the next protocol entity 1710 for processing in step 1560, that protocol entity 1710 retrieves the data from the buffer in the OR 1300 and deposits the processed result there in step 1570 when its process is complete.

When the data has been completely processed by the protocol entity 1710 currently processing it at step 1580, the session object repeats step 1550 to check whether more protocol entities are needed for the data. Should the data provided in the buffer stored in the OR 1300 end before the protocol is satisfied, the data is assumed to be incomplete at step 1580. If the data is incomplete, the protocol entity 1710 and the session object 240, 440 cannot complete their respective tasks, so another buffer is read

from the input channel, repeating step 1510, and the session object waits until further data for this session is sent to it by the session manager.

If no more protocol entities are needed at step 1560, the session object 240 uses the packers 720 corresponding to the protocol entities used by the session object to pack the data from the buffer in the OR 1300 into a serial stream of bytes, at step 1590. This CIP-formatted stream of bytes is transferred to the output queue 250 going to the routing manager. At this point, the processing cycle is complete, step 1600.

Referring to FIG. 17, the process of converting content data from CIP to application format is described and starts at step 1700. The protocol manager 2c and 4c reads data from the input queue 1410 and sends the data to the session manager 420, step 2710. Next, the session manager 420 sends the data buffer to one of the available session handlers 430, step 720. The available session handler checks whether the data belongs to an existing session or whether a new session needs to be created.

The session handler then sends the data to the appropriate session object 440, step 730. The session object 440 uses various unpackers 720 to unpack the CIP information included in the data and stores the individual data items in the OR 1300, step 740. The session object 440, at step 750, consults the PET 310 and information in the OR 1300 for the identity of the next Protocol Entity that should process the data which is now in the Object Repository 1300. If there is such a Protocol Entity 710, step 760, the control passes to it, and that Protocol Entity 1710 reconstruct its application layer data from the data in the OR 1300, step 770. At step 780, the Protocol Entity 710 determines if the operation is completed, upon which case execution resumes at step

750 by determining the next Protocol Entity. Otherwise, execution proceeds at the beginning, where more data is awaited, step 2710.

When all Protocol Entities are exhausted, step 760, the reconstructed data (which is deposited in the Object Repository 1310) is sent to the "Queue To Routing in Application" 450, and the process cycle is complete 790.

A sample PET 1310 is shown in Fig. 18. As seen in the drawing, the PET 1310 indicates which Protocol Entities are selected at given points, such as the start of the processing (when the TCP/IP Protocol entity is used), and thereafter. The PET 1310 also indicates what rules and conditions are required to trigger use of the given Protocol Entity. As explained above, the session handlers consult the PET 1310 to determine which Protocol Entities are to be employed at given stages in the conversion process.

The CIP and IRP Protocols

In accordance with the present invention, the security gateway's internal CIP and IRP protocols replace a message's native protocols in the link between the robots. In this way, data transfer is implemented only for specified data content within a given protocol. For instance, when a message using a particular application protocol is decomposed by the external robot the external robot's CIP translator for that protocol may encode only the image information that provides the application's GUI. Similarly, the internal robot's translator may implement CIP encoding only for the user's mouse and keyboard input responding to those graphics. In this instance, command codes are not passed.

The process that defines permissible sub-set of the syntax and the

functional suite of a given protocol that is to be allowed to pass into the trusted environment, as well as defining its representation in CIP, is carried out in several separate steps, occurring at different times in the robot processes. First, the user identifies the set of protocols or protocol characteristics that will be allowed to pass into the trusted environment. This can be conveniently done in a fourth generation language (4GL) referred to as protocol-definition language ("PeDaL"), which handles string literals, and provides a binary virtual machine language (VML) to replacing "C" as the target language.

Selected command codes may also be passed, by being either explicitly or implicitly coded by the translator. For example, a bill-viewing application can be secured in accordance with the present invention by generating a corresponding CIP protocol, using normative rules used by the messages native protocol and selecting from the native protocol only content that the network administrator of the secure environment deems to be secure from attacks that could use the bill-viewing application as vehicle for entering the trusted environment.

For the sake of simplicity, assume that only the following commands and flow sequences are defined by the bill-viewing protocol:

1) The "LOGIN" command should be issued, with "username" (8 characters exactly) and "password" (8 characters exactly) as its arguments; then
2) The user may choose to issue one of three commands: PRINT, VIEW or LOGOUT to either produce a printout of a bill, to view a bill on the screen, or to quit the application, respectively. However, when response to a PRINT or VIEW command has been completed, the system is ready to receive a new

command. In contrast, after receiving the LOGOUT command, the system resets to its initial state, and so, responds only to a LOGIN command thereafter. The corresponding CIP protocol format for command sequences compliant with the bill-viewer's native application protocol is as follows:

1) The unknown structured string of 16 bytes representing a username+password (first 8 bytes represent the 8 characters of the username, in ASCII code, and the last 8 bytes represent the password in ASCII code) pair come first in this format.

2) One of the known limited set of strings:, three commands: 0 and 3 for PRINT, 1 for VIEW and 2 for LOGOUT, is identified in the next two bits,

3) The commands PRINT (=0 or =3) and VIEW (=1) can be followed by any other command identifier., whereas

4) The command LOGOUT ends any sequence.

After an initial valid string of sixteen characters is received the interpreter responds in one of two ways: Any time a CIP interpreter receives a LOGOUT command, coded as "2", the interpreter resets to its initial state and awaits the next 16-byte sequence. If PRINT (=0 or =3) or VIEW (=1) appear instead, the interpreter will then respond to any of the three commands' codes. Since every sequence of bits must have a valid interpretation, having three commands represented by two bits makes one of the bit combinations superfluous. As still an interpretation is needed for the redundant combination, an arbitrary command is chosen to be represented also by this combination, in this example, the PRINT command.

Note that the LOGIN command is, in effect, a "constant" string. Therefore it is not explicitly coded by the CIP protocol. LOGIN can be implicitly passed and reinserted by the complementary interpreter in the other robot, because at the time when the sixteen character string is received, no command other than LOGIN can be processed by the interpreter, nor is LOGIN processed under any other circumstances. Therefore there is no point in explicitly providing a command-identifier value for LOGIN or other such "constant strings".

Numbers are accompanied by a logically associated "sanity range" for consistency checking, except for dates. Only a single date format is allowed from any one given CIP translator or interpreter. Similarly, unknown, unstructured strings are provided a "sanity check" value stating an expected length limit for the string. The elements of the string are preferably mapped to a sequential range of characters.

Thus this CIP coding process simultaneously clears application-level messages of suspect classes of data and verifies the integrity of the internal robot, because corruption of the translator operation will produce a sequence of (although meaningless) valid commands, which do not harm the internal zone.

The IRP is a simplified transfer protocol adapted for use in a point-to-point communication link such as between the internal and external robots. Since the communication is point-to-point, no routing information is needed in the transfer protocol. In one embodiment, the IRP consists of a header to the CIP data having twelve bytes, of which four are used as follows, with the remaining bytes being made available for reserved uses:

- the first byte is NULL;

- the second byte contains a packet ID, i.e., a number from 0 to 127, for which a static variable is used to track and increment assigned packet IDs; and
- the third and fourth bytes contain the length of the data in the CIP message.

Formal Verification

For a security software product to fulfill its role, it should be totally reliable, i.e., the product must be proven to show it complies with its specification. The unique combination of ART algorithm/technology, and the specific architectures described, i.e. strongly decoupling the two robots, with a single link between them, enables one to prove the whole security gateway by verifying the correctness of the internal robot only. This is so because according to the use of reduction methodology as described herein, anything sent on the communication bus has a valid interpretation by the internal robot, assuming it is verified, although that interpretation may be to meaningless data, as explained above. Furthermore, the separation of the security tasks into internal robot and external robot, keeping the internal robot as simple as possible by having all the "heavy" work, e.g. parsing protocols, carried out in the external robot, and furthermore using a simple point-to-point inter-robot protocol such as IRP to simplify the bus driver, makes verifying the internal robot a practical goal.

In order to formally verify the internal robot, it may be useful to take the top-down approach, assuming first that the robot modules (PM, RM, CM, App-Proxy, Web-Proxy) are verified, proving that their combination yields the desired properties (proper interpretation of the input to CM, i.e. the output from the Web-proxy and the application proxy should comply with the specifications provided by the owners of the internal applications.

The properties of each module can be expressed in terms of the output it produces, e.g., in the various output channels available to it - whether as memory area in a shared memory, or as I/O ports connected to peripheral equipment/devices, assuming the input it receives is in a preset format. This enables the verification of the overall property by means of an individual module verification in terms of I/O channels. Abstracting unnecessary information, such a system can be described by a specification language such as CSP or its derivatives, and be proven by such tools as SPIN and FDR2.

Therefore, verifying the internal robot amounts to verifying each module against its output channel properties, assuming the correctness of its input channels.

Then, one may proceed to proving each individual module, again by decomposing it to its sub-modules. The process of decomposition repeats itself down to the level of "atomic" code pieces, e.g. functions and procedures, where the decomposition can no longer be applied. However, these code pieces are usually small, therefore are verifiable by "direct" methods. These methods may include manual arguments, as well as mechanized methods such as theorem provers (NQTHM, ACL2, PVS) and model checkers (SPIN, STeP, etc.).

The invention has been described with particular reference to presently preferred embodiments thereof, but it will be apparent to one skilled in the art that variations and modification are possible within the spirit and scope of the invention.

WHAT IS CLAIMED IS:

1. A method for protecting an operating environment on a processor from a first program operating on the processor, the method comprising:

allocating memory space for use only by the first program while the first program is executing;

allowing communication between the first program and the operating environment through only a single link employing a single method selected from the group consisting of a shared memory space, a dedicated interrupt, and a dedicated I/O port; and

managing a restricted operating environment for the first program on the processor, the restricted operating environment preventing the first program from accessing resources on the processor except for the allocated memory space and the single communication link.

2. The method of claim 1, comprising executing the first program within the restricted operating environment.

3. The method of claim 2, comprising terminating execution of the first program upon an attempt by the first program to access a resource on the processor which is restricted by the restricted operating environment.

4. The method of claim 2, comprising ignoring any request by the first program to access a resource on the processor which is restricted by the restricted operating environment and attempting to continue execution of the first program without granting access to the restricted resource.

5. The method of claim 2, comprising allowing the operating environment to access the allocated memory space only prior to execution of the first program and after termination of first program execution.

6. The method of claim 5, comprising the operating environment writing data to the allocated memory space for use by the first program prior to execution of the first program, and the operating environment reading data generated by the first program from the allocated memory space after termination of first program execution.

7. The method of claim 1, wherein the step of allowing communication between the first program and the operating environment comprises allowing communication through only the shared memory space.

8. The method of claim 7, comprising the first program and operating environment each writing data to the shared memory space and indicating the availability of the data written to the shared memory space by setting a bit in a predefined location in the shared memory space.

9. The method of claim 8, comprising the operating environment and first program reading the data written to the shared memory space when the bit is set.

10. The method of claim 1, wherein the step of allowing communication between the first program and the operating environment comprises allowing communication through only the dedicated interrupt.

11. The method of claim 10, comprising handling the dedicated interrupt by moving data written by the first program to the allocated memory space to memory space of the operating environment or by moving data from memory space of the operating environment to the allocated memory space.

12. The method of claim 1, wherein the step of allowing communication between the first program and the operating environment comprises allowing communication through only the dedicated I/O port.

13. The method of claim 1, comprising, upon the occurrence of a hardware interrupt in the processor while the first program is executing, handling the hardware interrupt through an interrupt routine in the operating environment.

14. The method of claim 1, wherein the first program is an operating system.

15. A method for protecting an operating environment on a processor from a first program operating on the processor, the method comprising:

allocating memory space for use only by the first program while the first program is executing;

allowing the operating environment to access the allocated memory space only prior to execution of the first program and after termination of first program execution;

executing the first program; and

managing a restricted operating environment for the first program on the processor, the restricted operating environment preventing the first program from accessing resources on the processor except for the allocated memory space.

16. The method of claim 15, comprising the operating environment writing data to the allocated memory space for use by the first program prior to execution of the first program, and the operating environment reading data generated by

the first program from the allocated memory space after termination of first program execution.

17. A system for protecting an operating environment on a processor coupled to a memory device from a first program operating on the processor, the system comprising:

an allocated memory space in the memory device for use only by the first program while the first program is executing;

a communication link between the first program and the operating environment employing a single method selected from the group consisting of a shared memory space, a dedicated interrupt, and a dedicated I/O port; and

a mediator program for managing a restricted operating environment for the first program on the processor, the restricted operating environment preventing the first program from accessing resources on the processor except for the allocated memory space and the single communication link.

18. A method for operating a plurality of operating systems on a single processor, the method comprising:

allocating sets of resources on the processor, each set being available for use by only one of the plurality of operating systems;

allowing each of the operating systems to operate on the processor and access the set of resources available to the respective operating system;

upon an attempt by a first of the operating systems to access one or more resources outside the set of resources available to the first operating system, determining which set of resources contains the one or more resources attempted to be

accessed and determining a which second operating system of the other operating systems has such set available to it; and

handling such attempted access of one or more resources through the second operating system.

19.The method of claim 18, wherein the one or more resource attempted to be accessed by the first operating system is a hardware interrupt issued by the first operating system.

20.The method of claim 18, comprising handing a H/W interrupt invoked by a peripheral wherein the monitor determines which operating system services the interrupt

21.A system monitor for operating a plurality of operating systems on a single processor comprising:

means for allocating sets of resources on the processor, each set being available for use by only one of the plurality of operating systems;

means allowing each of the operating systems to operate on the processor and allowing access to the set of resources available to the respective operating system;

means for determining, upon an attempt by a first of the operating systems to access one or more resources outside the set of resources available to the first operating system, which set of resources contains the one or more resources attempted to be accessed and for determining which second operating system of the other operating systems has such set available to it; and

means for handling such attempted access of one or more resources through the second operating system.

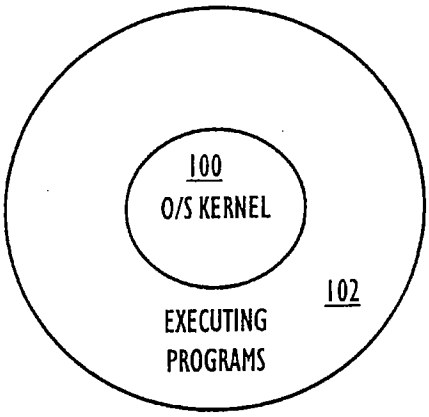



FIG. 1
(PRIOR ART)

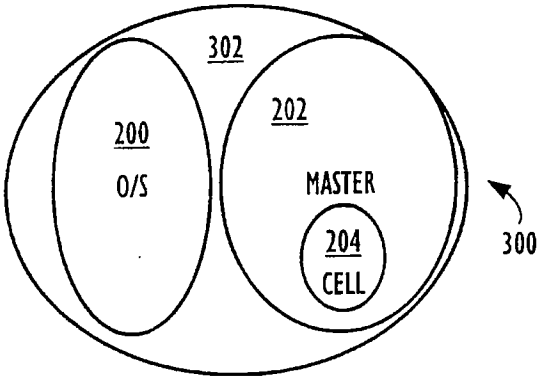



FIG. 2

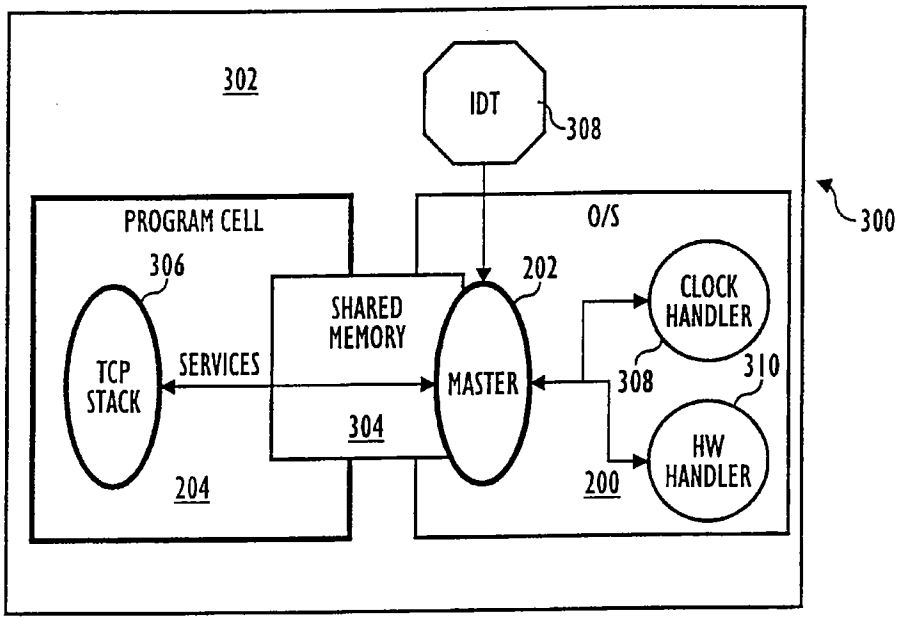

302
IDT
308
300
PROGRAM CELL
O/S
306
202
SHARED MEMORY
CLOCK HANDLER
TCP STACK
SERVICES
MASTER
308
310
304
HW HANDLER
204
200

FIG. 3

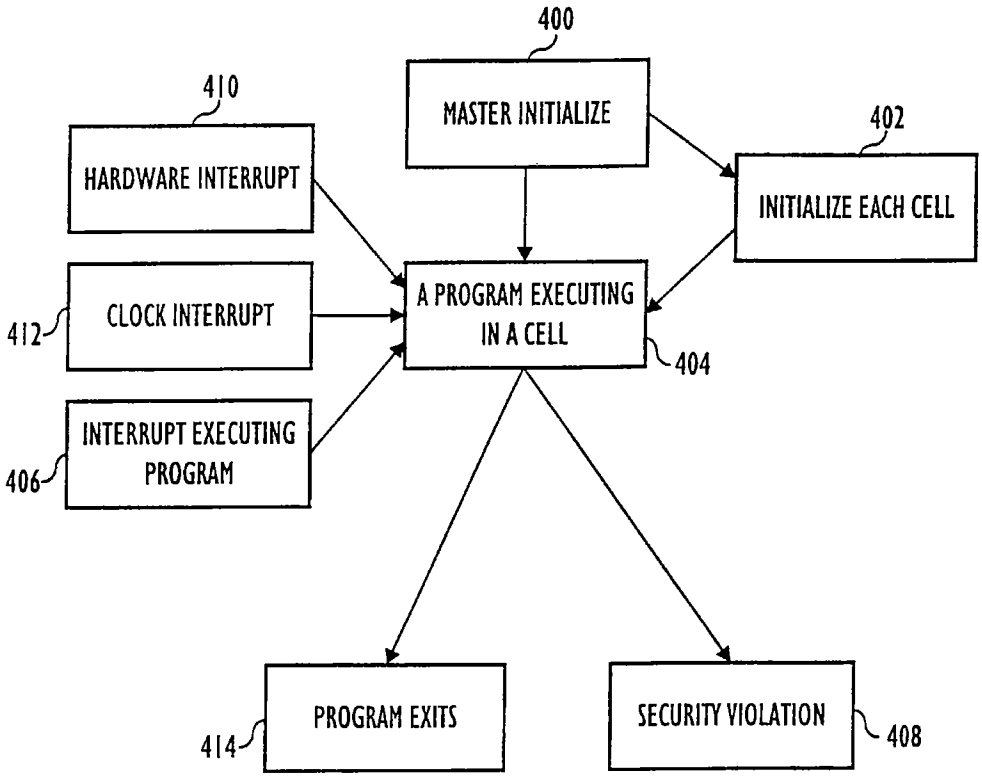

400
MASTER INITIALIZE
410
HARDWARE INTERRUPT
402
INITIALIZE EACH CELL
CLOCK INTERRUPT
412
A PROGRAM EXECUTING IN A CELL
404
INTERRUPT EXECUTING PROGRAM
406
PROGRAM EXITS
414
SECURITY VIOLATION
408

FIG. 4

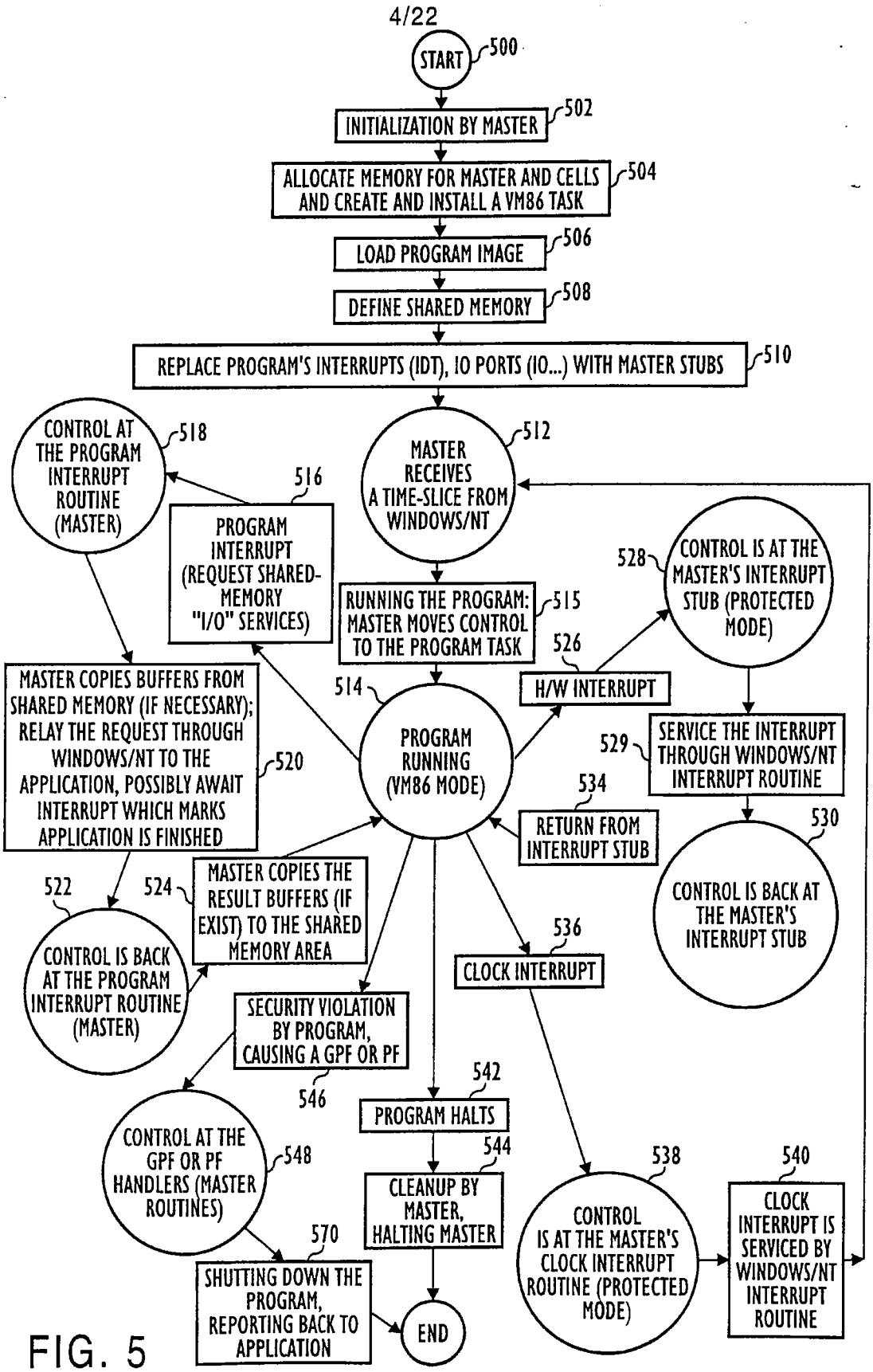

START
500
INITIALIZATION BY MASTER
502
ALLOCATE MEMORY FOR MASTER AND CELLS AND CREATE AND INSTALL A VM86 TASK
504
LOAD PROGRAM IMAGE
506
DEFINE SHARED MEMORY
508
REPLACE PROGRAM'S INTERRUPTS (IDT), IO PORTS (IO...) WITH MASTER STUBS
510
MASTER RECEIVES A TIME-SLICE FROM WINDOWS/NT
512
RUNNING THE PROGRAM: MASTER MOVES CONTROL TO THE PROGRAM TASK
515
PROGRAM RUNNING (VM86 MODE)
514
H/W INTERRUPT
526
CONTROL IS AT THE MASTER'S INTERRUPT STUB (PROTECTED MODE)
528
SERVICE THE INTERRUPT THROUGH WINDOWS/NT INTERRUPT ROUTINE
529
CONTROL IS BACK AT THE MASTER'S INTERRUPT STUB
530
RETURN FROM INTERRUPT STUB
534
CONTROL AT THE PROGRAM INTERRUPT ROUTINE (MASTER)
518
PROGRAM INTERRUPT (REQUEST SHARED-MEMORY "I/O" SERVICES)
516
MASTER COPIES BUFFERS FROM SHARED MEMORY (IF NECESSARY); RELAY THE REQUEST THROUGH WINDOWS/NT TO THE APPLICATION, POSSIBLY AWAIT INTERRUPT WHICH MARKS APPLICATION IS FINISHED
520
MASTER COPIES THE RESULT BUFFERS (IF EXIST) TO THE SHARED MEMORY AREA
524
CONTROL IS BACK AT THE PROGRAM INTERRUPT ROUTINE (MASTER)
522
CLOCK INTERRUPT
536
SECURITY VIOLATION BY PROGRAM, CAUSING A GPF OR PF
546
PROGRAM HALTS
542
CONTROL AT THE GPF OR PF HANDLERS (MASTER ROUTINES)
548
CLEANUP BY MASTER, HALTING MASTER
544
CONTROL IS AT THE MASTER'S CLOCK INTERRUPT ROUTINE (PROTECTED MODE)
538
CLOCK INTERRUPT IS SERVICED BY WINDOWS/NT INTERRUPT ROUTINE
540
SHUTTING DOWN THE PROGRAM, REPORTING BACK TO APPLICATION
570
END

FIG. 5

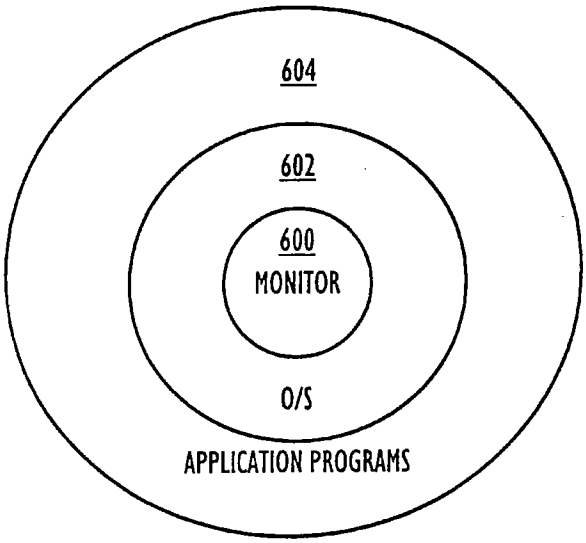



FIG. 6

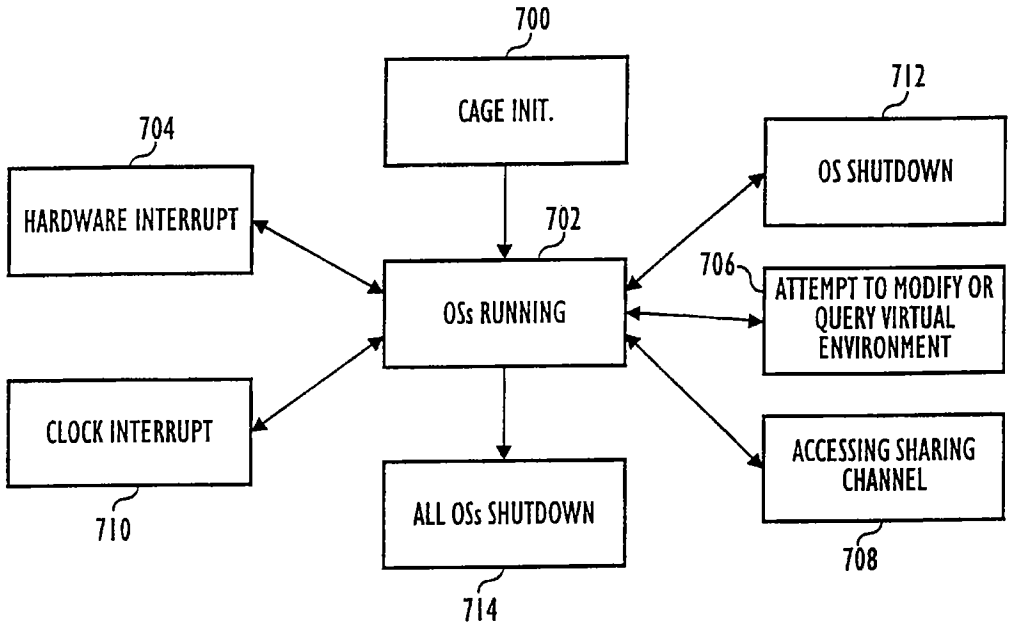

700
CAGE INIT.
712
OS SHUTDOWN
704
HARDWARE INTERRUPT
702
706
OSs RUNNING
ATTEMPT TO MODIFY OR QUERY VIRTUAL ENVIRONMENT
CLOCK INTERRUPT
ACCESSING SHARING CHANNEL
ALL OSs SHUTDOWN
710
708
714

FIG. 7

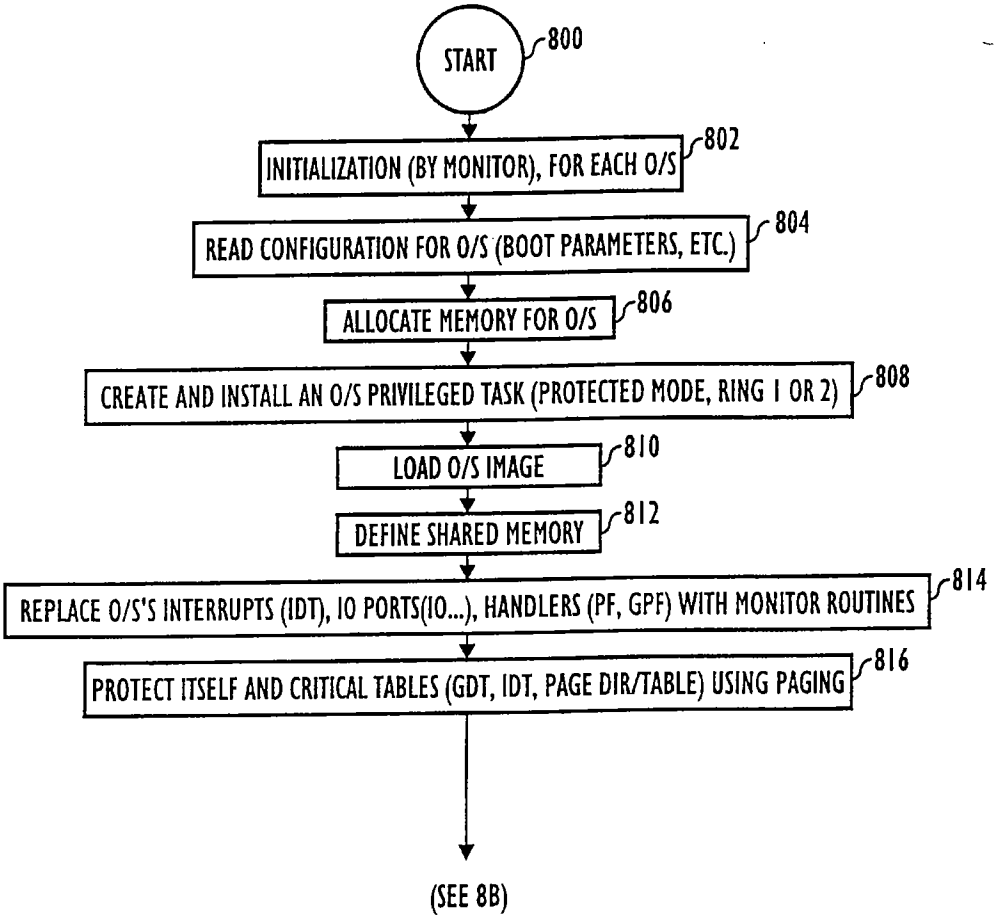

START
800
INITIALIZATION (BY MONITOR), FOR EACH O/S
802
READ CONFIGURATION FOR O/S (BOOT PARAMETERS, ETC.)
804
ALLOCATE MEMORY FOR O/S
806
CREATE AND INSTALL AN O/S PRIVILEGED TASK (PROTECTED MODE, RING 1 OR 2)
808
LOAD O/S IMAGE
810
DEFINE SHARED MEMORY
812
REPLACE O/S'S INTERRUPTS (IDT), IO PORTS(IO...), HANDLERS (PF, GPF) WITH MONITOR ROUTINES
814
PROTECT ITSELF AND CRITICAL TABLES (GDT, IDT, PAGE DIR/TABLE) USING PAGING
816
(SEE 8B)

FIG. 8A

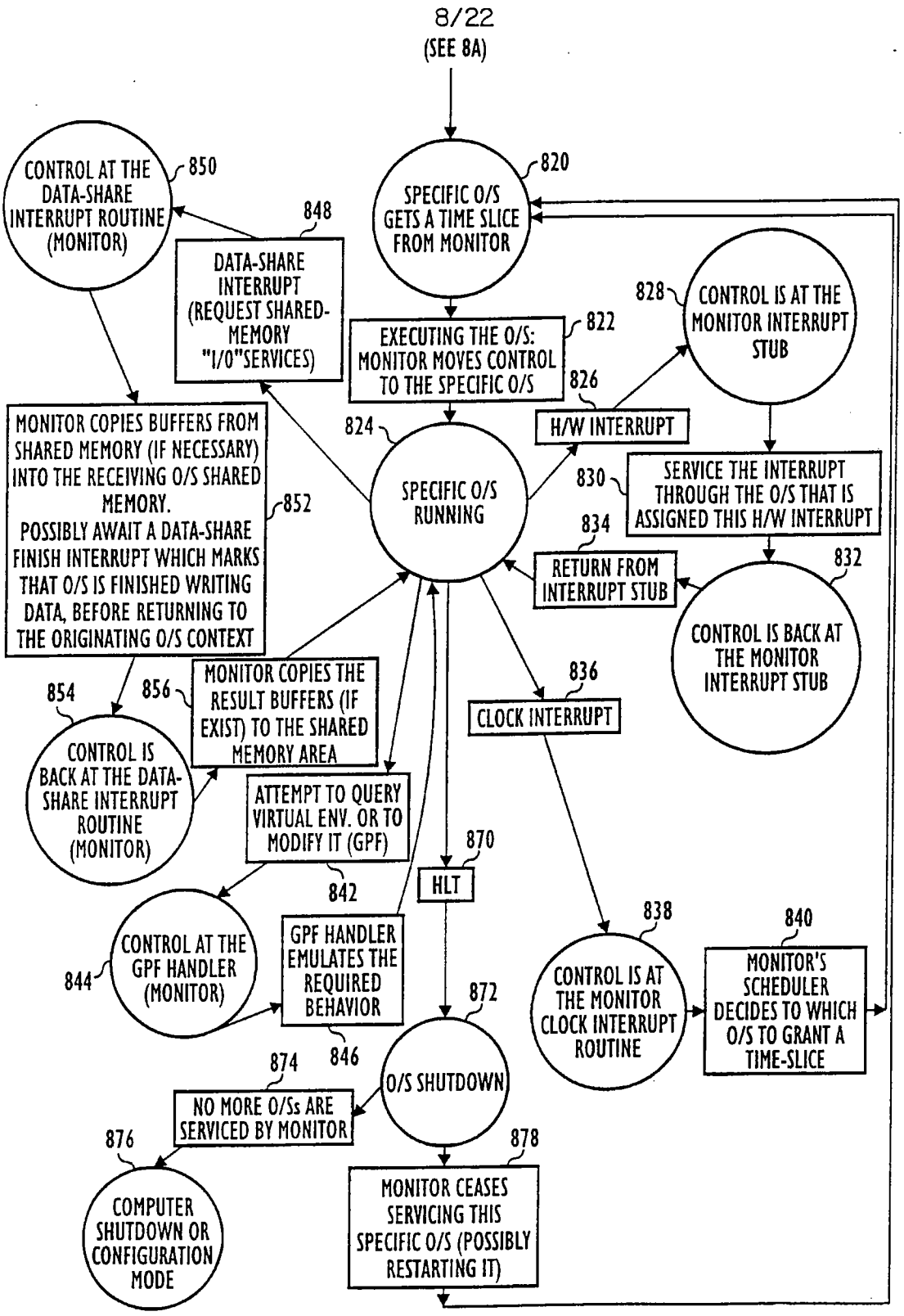

8/22
(SEE 8A)
CONTROL AT THE DATA-SHARE INTERRUPT ROUTINE (MONITOR)
850
848
DATA-SHARE INTERRUPT (REQUEST SHARED-MEMORY "I/O"SERVICES)
SPECIFIC O/S GETS A TIME SLICE FROM MONITOR
820
828
CONTROL IS AT THE MONITOR INTERRUPT STUB
EXECUTING THE O/S: MONITOR MOVES CONTROL TO THE SPECIFIC O/S
822
826
H/W INTERRUPT
MONITOR COPIES BUFFERS FROM SHARED MEMORY (IF NECESSARY) INTO THE RECEIVING O/S SHARED MEMORY.
POSSIBLY AWAIT A DATA-SHARE FINISH INTERRUPT WHICH MARKS THAT O/S IS FINISHED WRITING DATA, BEFORE RETURNING TO THE ORIGINATING O/S CONTEXT
852
824
SPECIFIC O/S RUNNING
830
SERVICE THE INTERRUPT THROUGH THE O/S THAT IS ASSIGNED THIS H/W INTERRUPT
834
RETURN FROM INTERRUPT STUB
832
CONTROL IS BACK AT THE MONITOR INTERRUPT STUB
854
856
MONITOR COPIES THE RESULT BUFFERS (IF EXIST) TO THE SHARED MEMORY AREA
836
CLOCK INTERRUPT
CONTROL IS BACK AT THE DATA-SHARE INTERRUPT ROUTINE (MONITOR)
ATTEMPT TO QUERY VIRTUAL ENV. OR TO MODIFY IT (GPF)
842
870
HLT
844
CONTROL AT THE GPF HANDLER (MONITOR)
GPF HANDLER EMULATES THE REQUIRED BEHAVIOR
838
CONTROL IS AT THE MONITOR CLOCK INTERRUPT ROUTINE
840
MONITOR'S SCHEDULER DECIDES TO WHICH O/S TO GRANT A TIME-SLICE
872
846
874
O/S SHUTDOWN
NO MORE O/Ss ARE SERVICED BY MONITOR
876
878
COMPUTER SHUTDOWN OR CONFIGURATION MODE
MONITOR CEASES SERVICING THIS SPECIFIC O/S (POSSIBLY RESTARTING IT)

FIG. 8B

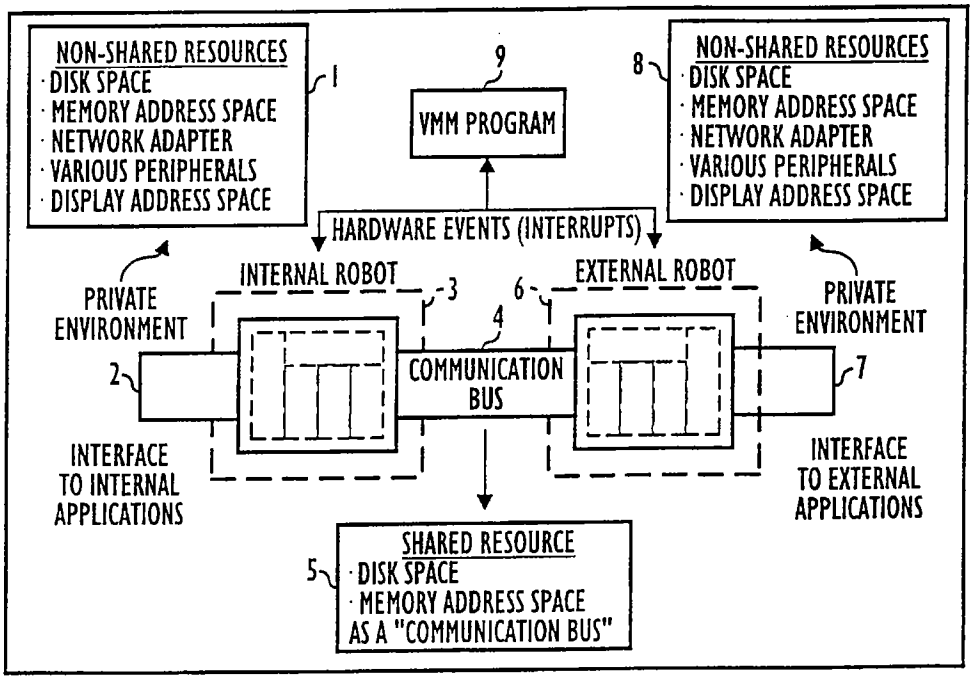

NON-SHARED RESOURCES
· DISK SPACE
· MEMORY ADDRESS SPACE
· NETWORK ADAPTER
· VARIOUS PERIPHERALS
· DISPLAY ADDRESS SPACE
1
9
VMM PROGRAM
8
NON-SHARED RESOURCES
· DISK SPACE
· MEMORY ADDRESS SPACE
· NETWORK ADAPTER
· VARIOUS PERIPHERALS
· DISPLAY ADDRESS SPACE
HARDWARE EVENTS (INTERRUPTS)
PRIVATE ENVIRONMENT
INTERNAL ROBOT
3
6
EXTERNAL ROBOT
PRIVATE ENVIRONMENT
4
2
COMMUNICATION BUS
7
INTERFACE TO INTERNAL APPLICATIONS
INTERFACE TO EXTERNAL APPLICATIONS
SHARED RESOURCE
5
· DISK SPACE
· MEMORY ADDRESS SPACE AS A "COMMUNICATION BUS"

FIG. 9

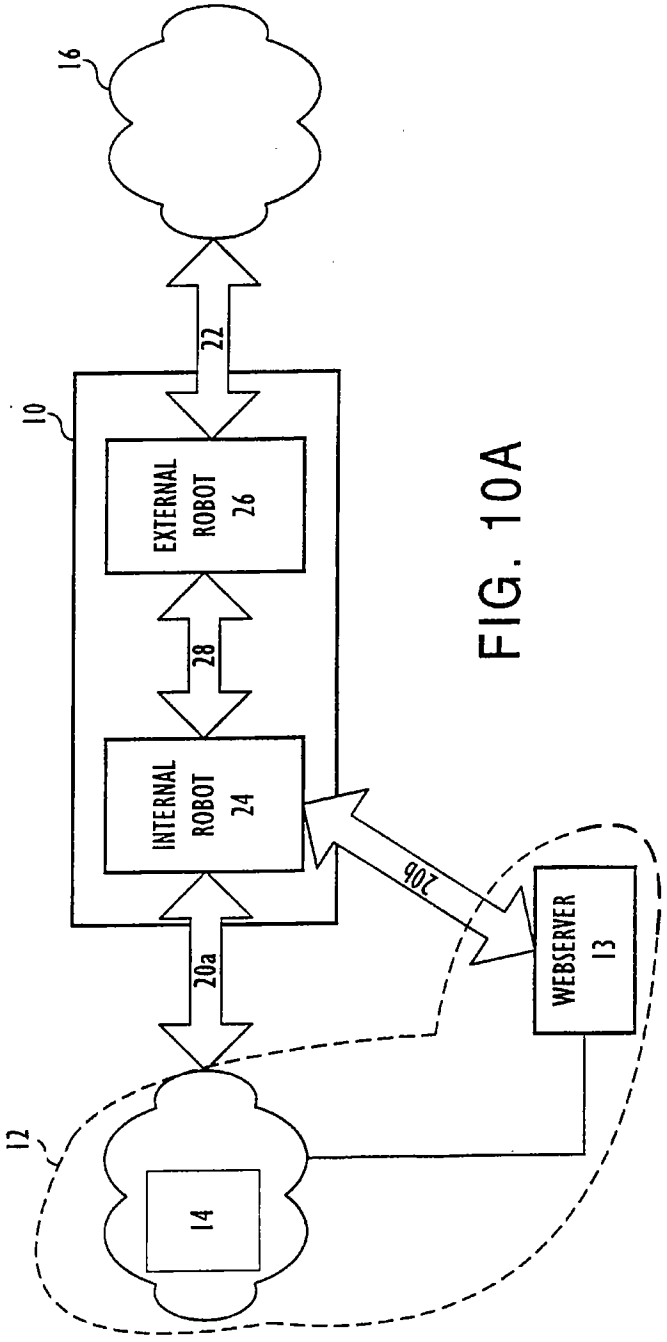

16
22
10
EXTERNAL ROBOT 26
28
INTERNAL ROBOT 24
20b
WEBSERVER 13
20a
12
14

FIG. 10A

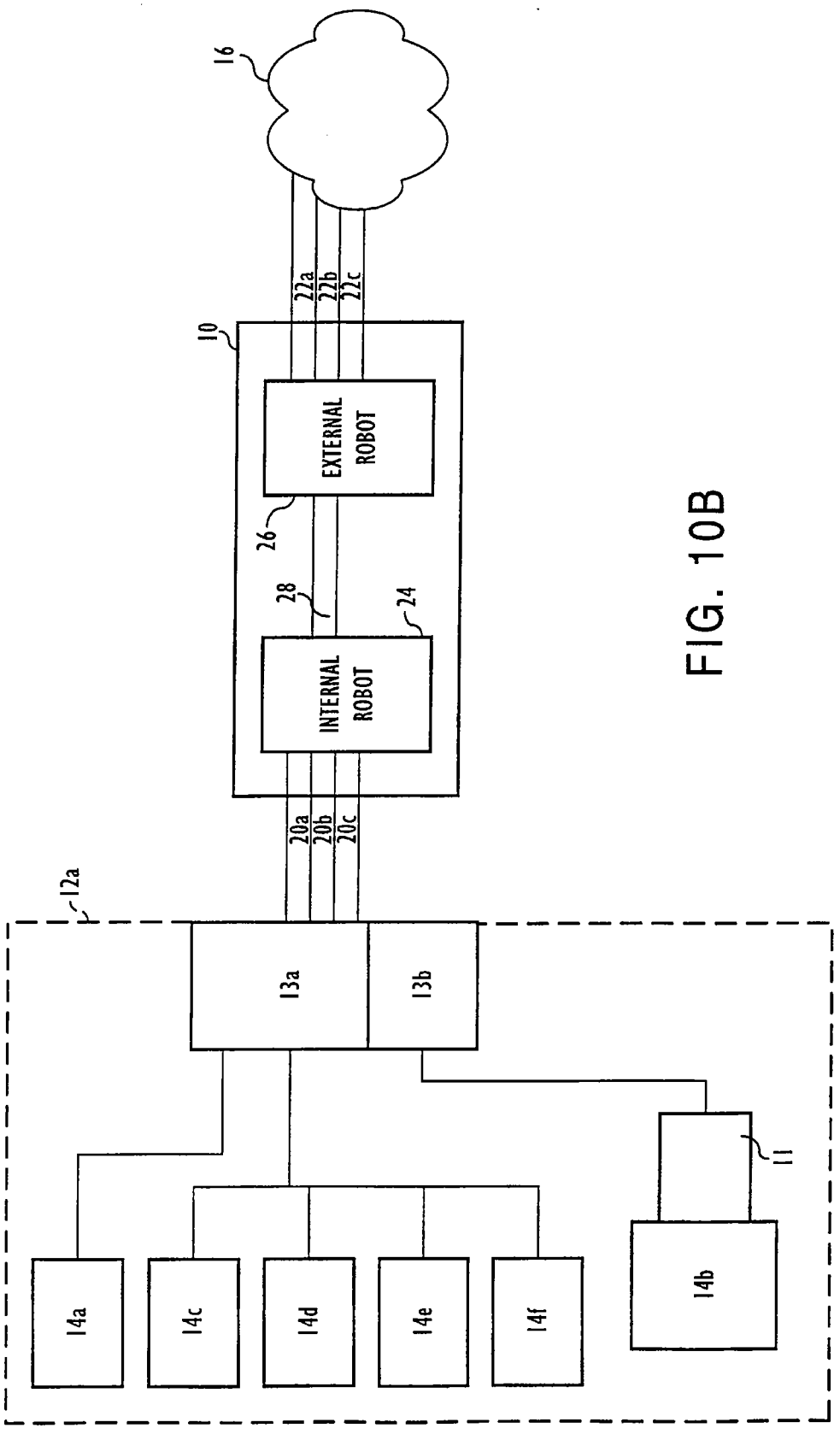

16
22a
22b
22c
10
EXTERNAL ROBOT
26
28
24
INTERNAL ROBOT
20a
20b
20c
12a
13a
13b
11
14a
14c
14d
14e
14f
14b

FIG. 10B

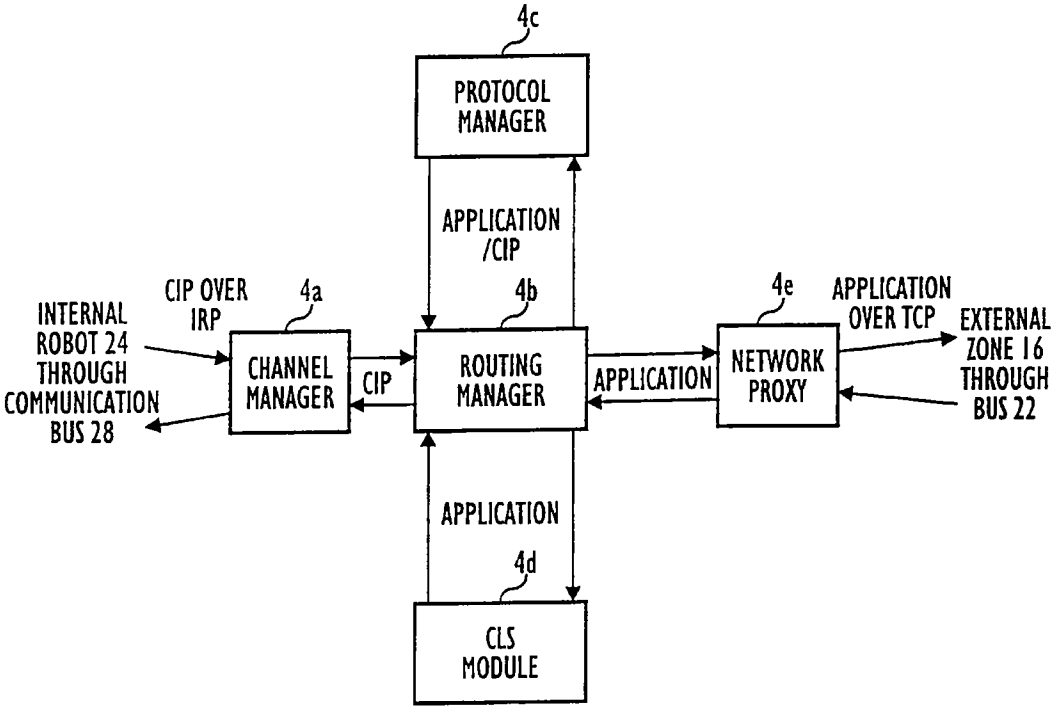

4c
PROTOCOL MANAGER
APPLICATION /CIP
CIP OVER IRP
4a
4b
4e
APPLICATION OVER TCP
INTERNAL ROBOT 24 THROUGH COMMUNICATION BUS 28
CHANNEL MANAGER
CIP
ROUTING MANAGER
APPLICATION
NETWORK PROXY
EXTERNAL ZONE 16 THROUGH BUS 22
APPLICATION
4d
CLS MODULE

FIG. 11A

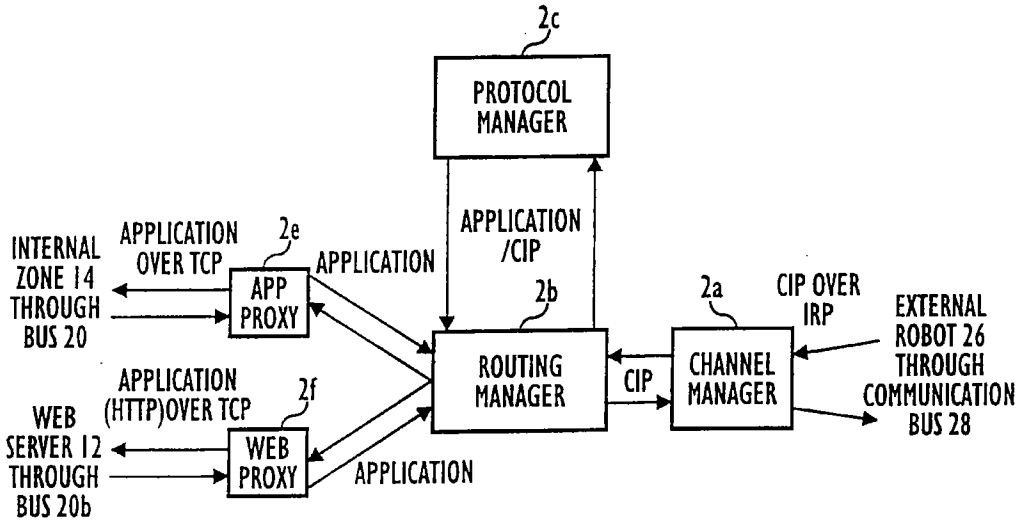

2c
PROTOCOL MANAGER
APPLICATION /CIP
INTERNAL ZONE 14 THROUGH BUS 20
APPLICATION OVER TCP
2e
APP PROXY
APPLICATION
2b
2a
CIP OVER IRP
EXTERNAL ROBOT 26 THROUGH COMMUNICATION BUS 28
ROUTING MANAGER
CIP
CHANNEL MANAGER
APPLICATION (HTTP)OVER TCP
2f
WEB SERVER 12 THROUGH BUS 20b
WEB PROXY
APPLICATION

FIG. 11B

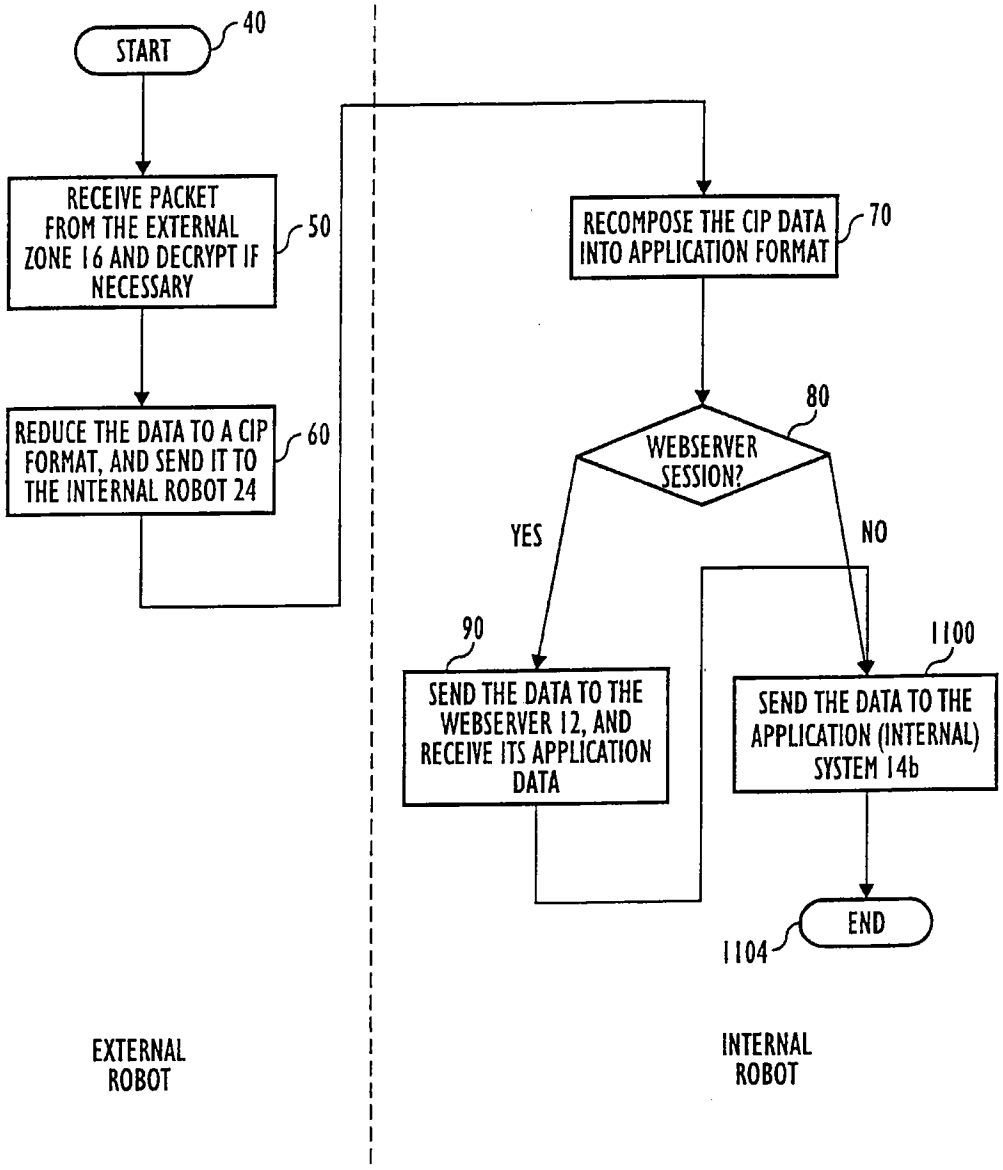

START
40
RECEIVE PACKET FROM THE EXTERNAL ZONE 16 AND DECRYPT IF NECESSARY
50
REDUCE THE DATA TO A CIP FORMAT, AND SEND IT TO THE INTERNAL ROBOT 24
60
RECOMPOSE THE CIP DATA INTO APPLICATION FORMAT
70
WEBSERVER SESSION?
80
YES
NO
SEND THE DATA TO THE WEBSERVER 12, AND RECEIVE ITS APPLICATION DATA
90
SEND THE DATA TO THE APPLICATION (INTERNAL) SYSTEM 14b
1100
END
1104
EXTERNAL ROBOT
INTERNAL ROBOT

FIG. 12A

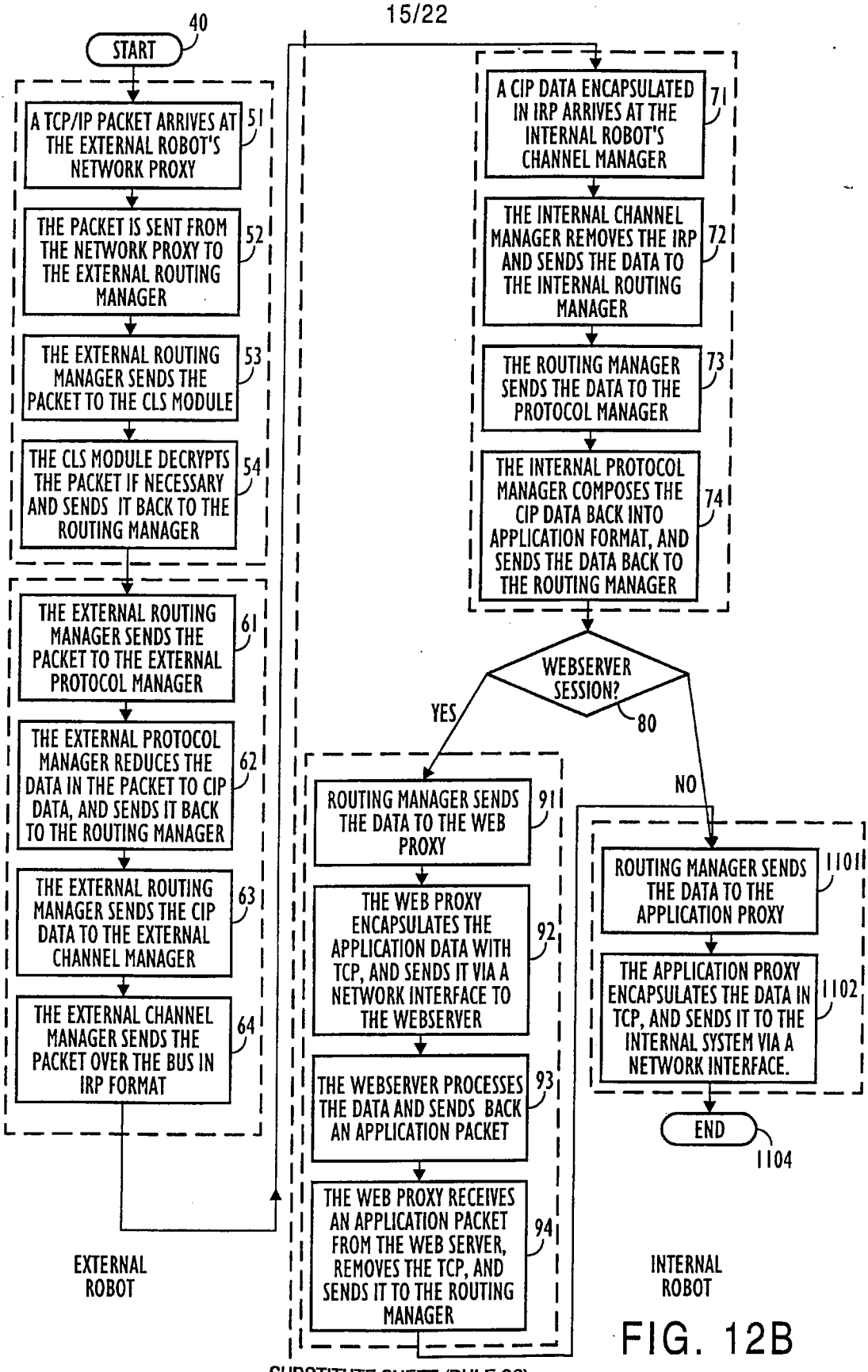

40
START
A TCP/IP PACKET ARRIVES AT THE EXTERNAL ROBOT'S NETWORK PROXY
51
THE PACKET IS SENT FROM THE NETWORK PROXY TO THE EXTERNAL ROUTING MANAGER
52
THE EXTERNAL ROUTING MANAGER SENDS THE PACKET TO THE CLS MODULE
53
THE CLS MODULE DECRYPTS THE PACKET IF NECESSARY AND SENDS IT BACK TO THE ROUTING MANAGER
54
THE EXTERNAL ROUTING MANAGER SENDS THE PACKET TO THE EXTERNAL PROTOCOL MANAGER
61
THE EXTERNAL PROTOCOL MANAGER REDUCES THE DATA IN THE PACKET TO CIP DATA, AND SENDS IT BACK TO THE ROUTING MANAGER
62
THE EXTERNAL ROUTING MANAGER SENDS THE CIP DATA TO THE EXTERNAL CHANNEL MANAGER
63
THE EXTERNAL CHANNEL MANAGER SENDS THE PACKET OVER THE BUS IN IRP FORMAT
64
EXTERNAL ROBOT
A CIP DATA ENCAPSULATED IN IRP ARRIVES AT THE INTERNAL ROBOT'S CHANNEL MANAGER
71
THE INTERNAL CHANNEL MANAGER REMOVES THE IRP AND SENDS THE DATA TO THE INTERNAL ROUTING MANAGER
72
THE ROUTING MANAGER SENDS THE DATA TO THE PROTOCOL MANAGER
73
THE INTERNAL PROTOCOL MANAGER COMPOSES THE CIP DATA BACK INTO APPLICATION FORMAT, AND SENDS THE DATA BACK TO THE ROUTING MANAGER
74
WEBSERVER SESSION?
80
YES
NO
ROUTING MANAGER SENDS THE DATA TO THE WEB PROXY
91
THE WEB PROXY ENCAPSULATES THE APPLICATION DATA WITH TCP, AND SENDS IT VIA A NETWORK INTERFACE TO THE WEBSERVER
92
THE WEBSERVER PROCESSES THE DATA AND SENDS BACK AN APPLICATION PACKET
93
THE WEB PROXY RECEIVES AN APPLICATION PACKET FROM THE WEB SERVER, REMOVES THE TCP, AND SENDS IT TO THE ROUTING MANAGER
94
ROUTING MANAGER SENDS THE DATA TO THE APPLICATION PROXY
1101
THE APPLICATION PROXY ENCAPSULATES THE DATA IN TCP, AND SENDS IT TO THE INTERNAL SYSTEM VIA A NETWORK INTERFACE.
1102
END
1104
INTERNAL ROBOT

FIG. 12B

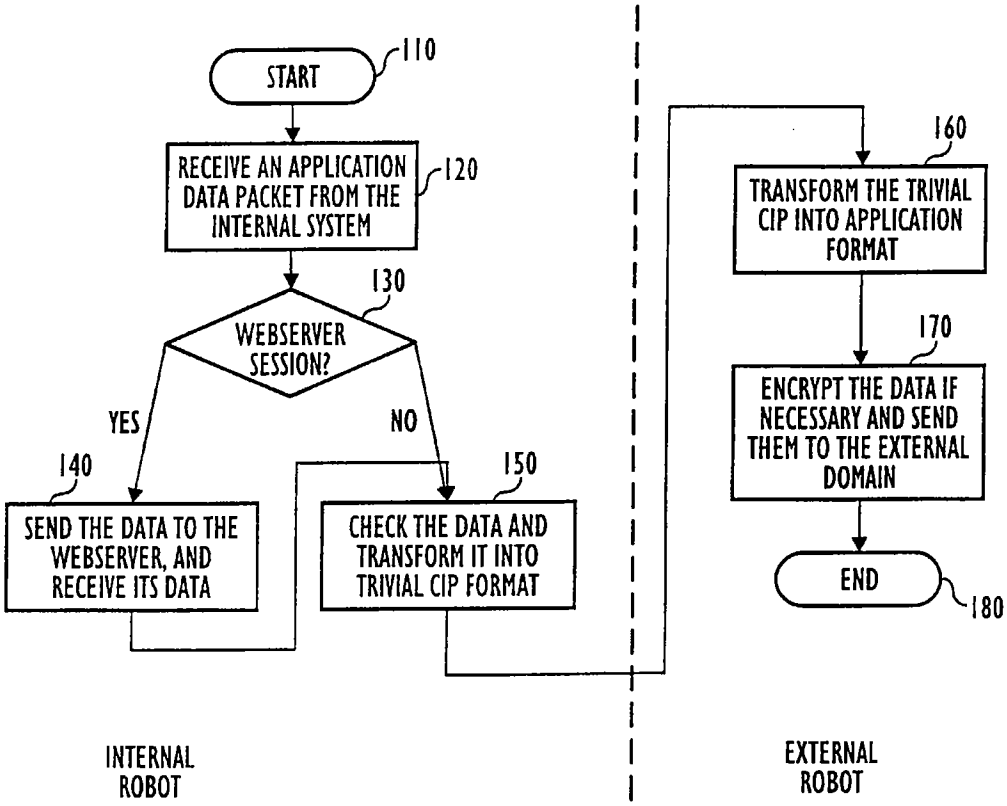

START
110
RECEIVE AN APPLICATION DATA PACKET FROM THE INTERNAL SYSTEM
120
WEBSERVER SESSION?
130
YES
NO
140
SEND THE DATA TO THE WEBSERVER, AND RECEIVE ITS DATA
150
CHECK THE DATA AND TRANSFORM IT INTO TRIVIAL CIP FORMAT
160
TRANSFORM THE TRIVIAL CIP INTO APPLICATION FORMAT
170
ENCRYPT THE DATA IF NECESSARY AND SEND THEM TO THE EXTERNAL DOMAIN
END
180
INTERNAL ROBOT
EXTERNAL ROBOT

FIG. 13A

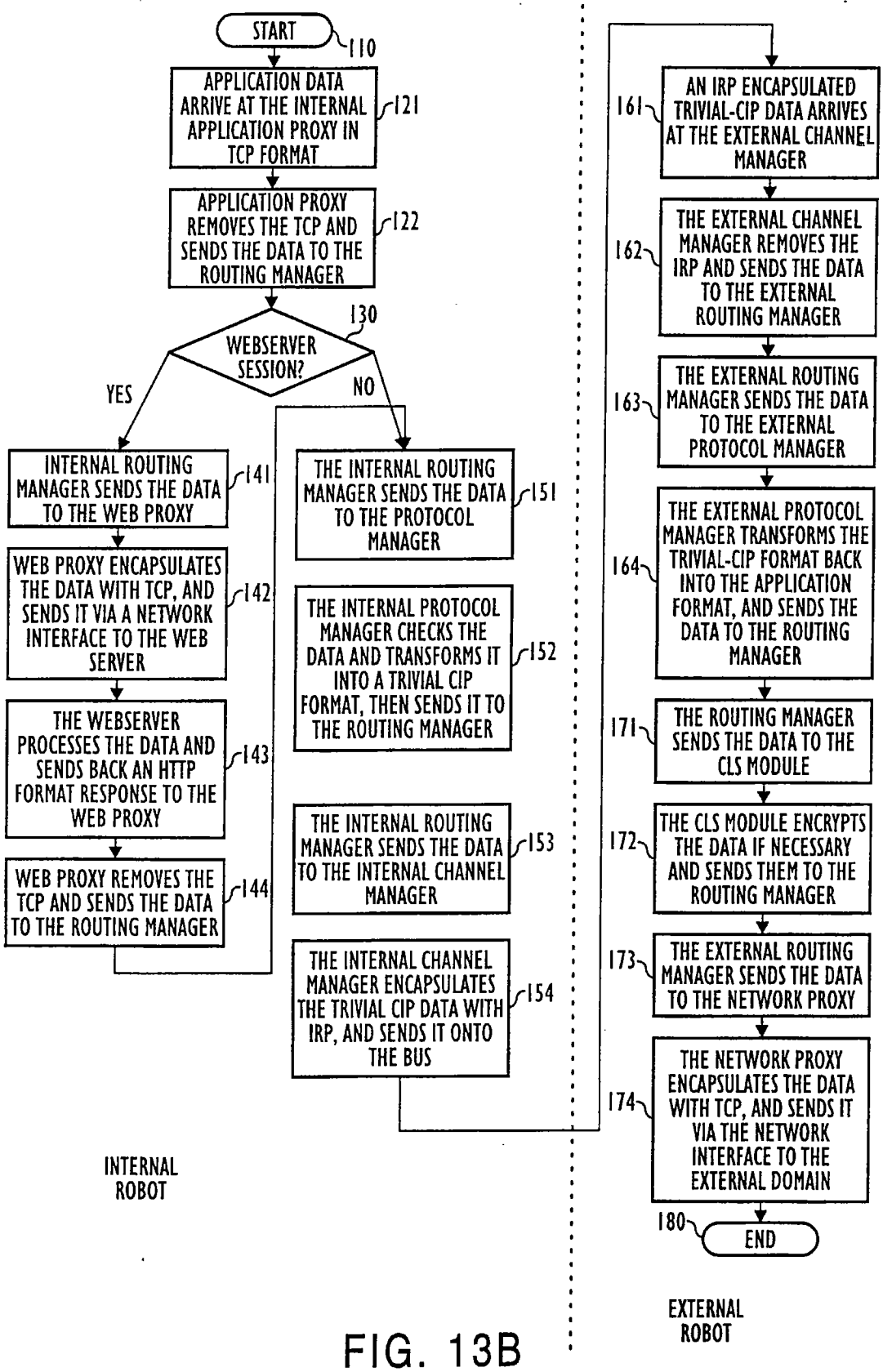

START
110
APPLICATION DATA ARRIVE AT THE INTERNAL APPLICATION PROXY IN TCP FORMAT
121
APPLICATION PROXY REMOVES THE TCP AND SENDS THE DATA TO THE ROUTING MANAGER
122
WEBSERVER SESSION?
130
YES
NO
INTERNAL ROUTING MANAGER SENDS THE DATA TO THE WEB PROXY
141
WEB PROXY ENCAPSULATES THE DATA WITH TCP, AND SENDS IT VIA A NETWORK INTERFACE TO THE WEB SERVER
142
THE WEBSERVER PROCESSES THE DATA AND SENDS BACK AN HTTP FORMAT RESPONSE TO THE WEB PROXY
143
WEB PROXY REMOVES THE TCP AND SENDS THE DATA TO THE ROUTING MANAGER
144
THE INTERNAL ROUTING MANAGER SENDS THE DATA TO THE PROTOCOL MANAGER
151
THE INTERNAL PROTOCOL MANAGER CHECKS THE DATA AND TRANSFORMS IT INTO A TRIVIAL CIP FORMAT, THEN SENDS IT TO THE ROUTING MANAGER
152
THE INTERNAL ROUTING MANAGER SENDS THE DATA TO THE INTERNAL CHANNEL MANAGER
153
THE INTERNAL CHANNEL MANAGER ENCAPSULATES THE TRIVIAL CIP DATA WITH IRP, AND SENDS IT ONTO THE BUS
154
INTERNAL ROBOT
AN IRP ENCAPSULATED TRIVIAL-CIP DATA ARRIVES AT THE EXTERNAL CHANNEL MANAGER
161
THE EXTERNAL CHANNEL MANAGER REMOVES THE IRP AND SENDS THE DATA TO THE EXTERNAL ROUTING MANAGER
162
THE EXTERNAL ROUTING MANAGER SENDS THE DATA TO THE EXTERNAL PROTOCOL MANAGER
163
THE EXTERNAL PROTOCOL MANAGER TRANSFORMS THE TRIVIAL-CIP FORMAT BACK INTO THE APPLICATION FORMAT, AND SENDS THE DATA TO THE ROUTING MANAGER
164
THE ROUTING MANAGER SENDS THE DATA TO THE CLS MODULE
171
THE CLS MODULE ENCRYPTS THE DATA IF NECESSARY AND SENDS THEM TO THE ROUTING MANAGER
172
THE EXTERNAL ROUTING MANAGER SENDS THE DATA TO THE NETWORK PROXY
173
THE NETWORK PROXY ENCAPSULATES THE DATA WITH TCP, AND SENDS IT VIA THE NETWORK INTERFACE TO THE EXTERNAL DOMAIN
174
180
END
EXTERNAL ROBOT

FIG. 13B

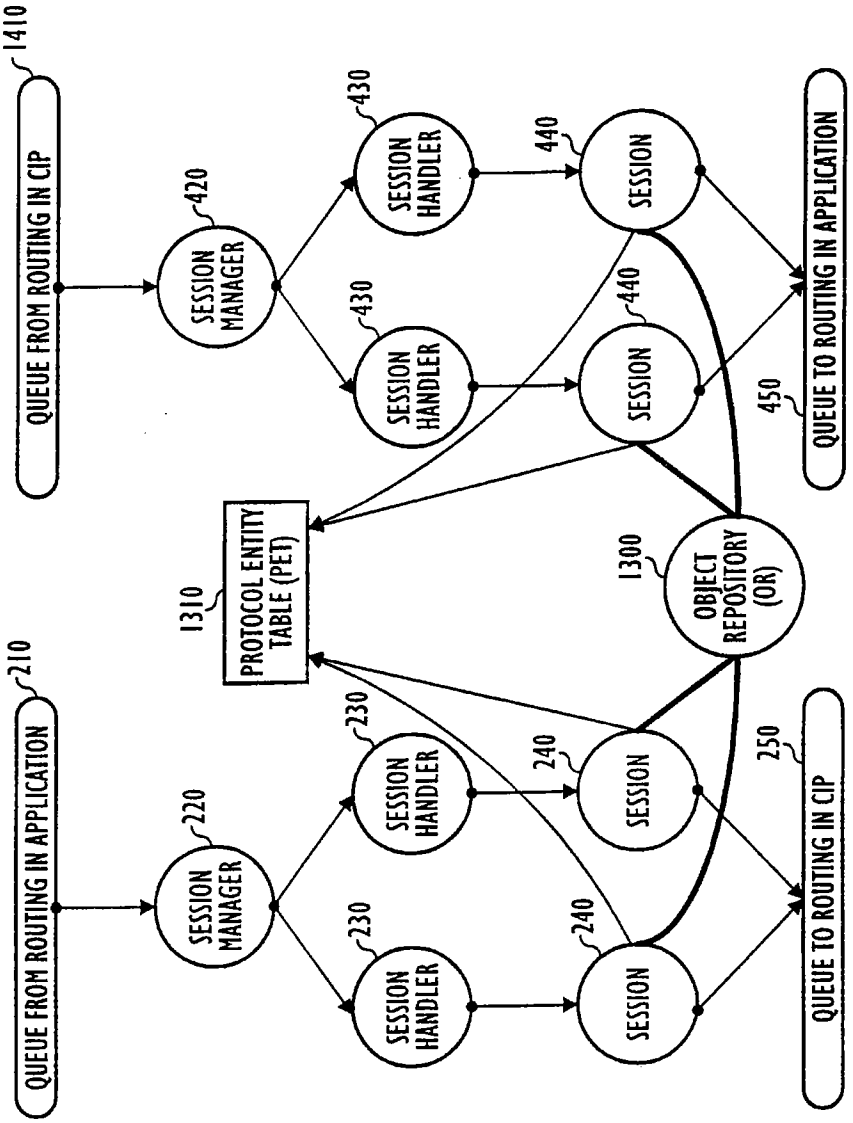

QUEUE FROM ROUTING IN APPLICATION
210
SESSION MANAGER
220
SESSION HANDLER
230
SESSION HANDLER
230
SESSION
240
SESSION
240
QUEUE TO ROUTING IN CIP
250
PROTOCOL ENTITY TABLE (PET)
1310
OBJECT REPOSITORY (OR)
1300
QUEUE FROM ROUTING IN CIP
1410
SESSION MANAGER
420
SESSION HANDLER
430
SESSION HANDLER
430
SESSION
440
SESSION
440
QUEUE TO ROUTING IN APPLICATION
450

FIG. 14

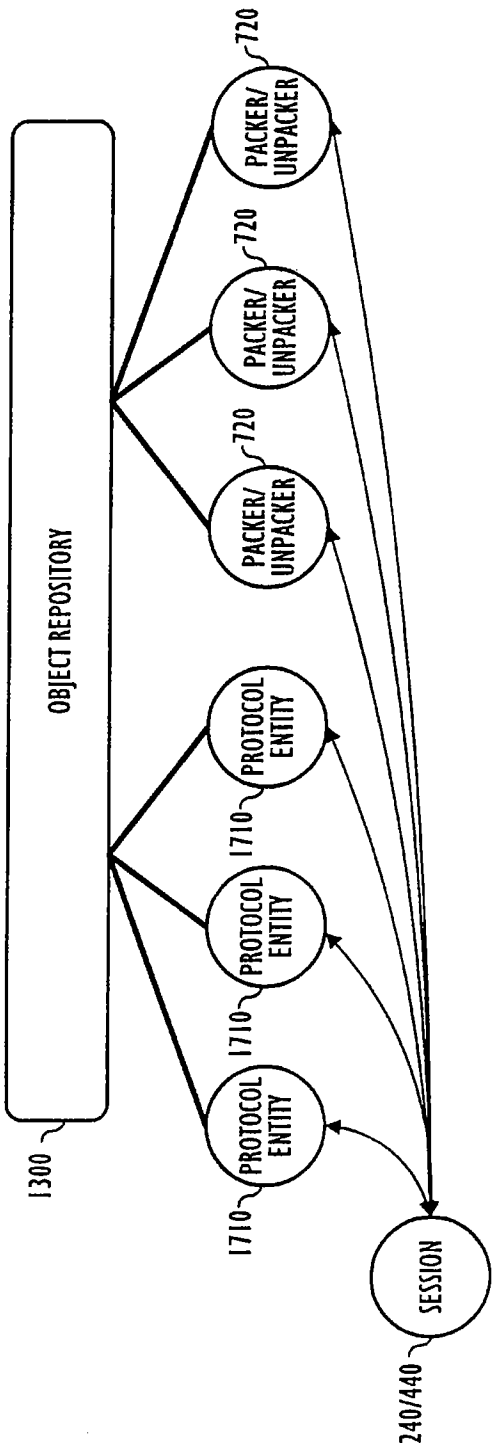

OBJECT REPOSITORY
1300
PACKER/ UNPACKER
720
PACKER/ UNPACKER
720
PACKER/ UNPACKER
720
PROTOCOL ENTITY
1710
PROTOCOL ENTITY
1710
PROTOCOL ENTITY
1710
SESSION
240/440

FIG. 15

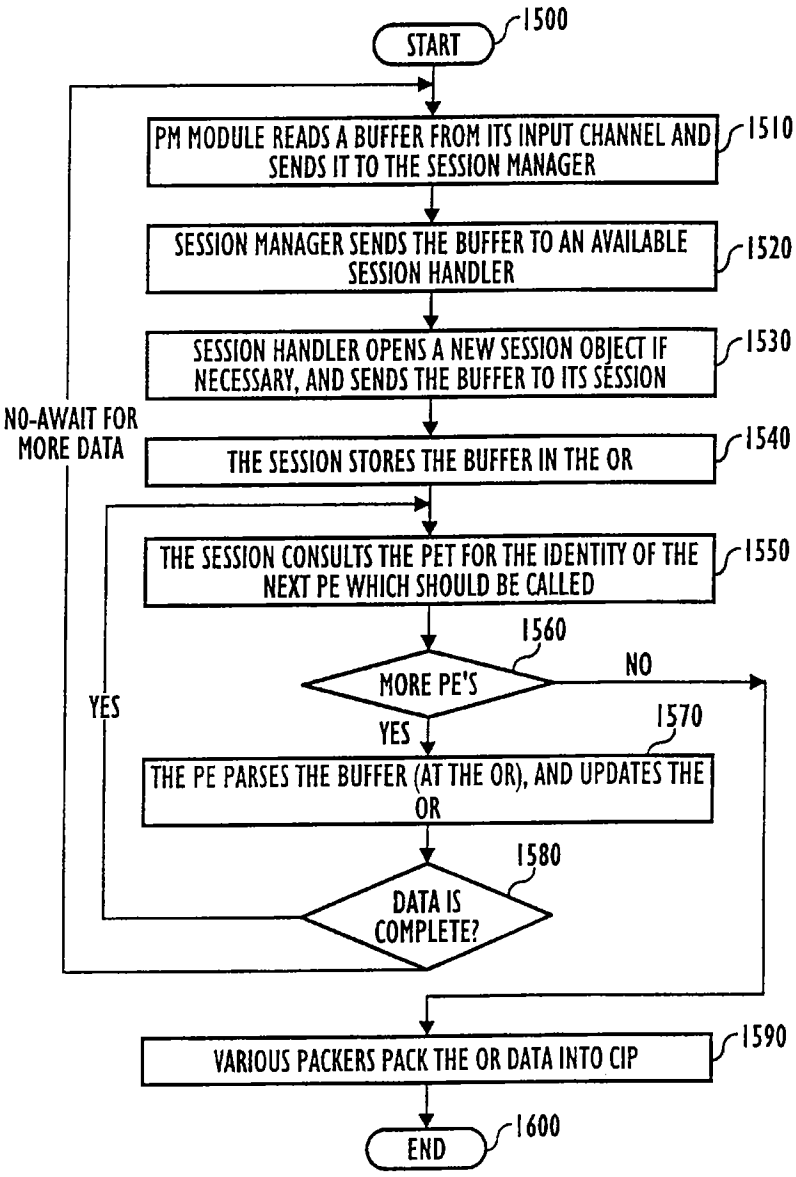

START
1500
PM MODULE READS A BUFFER FROM ITS INPUT CHANNEL AND SENDS IT TO THE SESSION MANAGER
1510
SESSION MANAGER SENDS THE BUFFER TO AN AVAILABLE SESSION HANDLER
1520
SESSION HANDLER OPENS A NEW SESSION OBJECT IF NECESSARY, AND SENDS THE BUFFER TO ITS SESSION
1530
NO-AWAIT FOR MORE DATA
THE SESSION STORES THE BUFFER IN THE OR
1540
THE SESSION CONSULTS THE PET FOR THE IDENTITY OF THE NEXT PE WHICH SHOULD BE CALLED
1550
1560
MORE PE'S
NO
YES
YES
1570
THE PE PARSES THE BUFFER (AT THE OR), AND UPDATES THE OR
1580
DATA IS COMPLETE?
VARIOUS PACKERS PACK THE OR DATA INTO CIP
1590
END
1600

FIG. 16

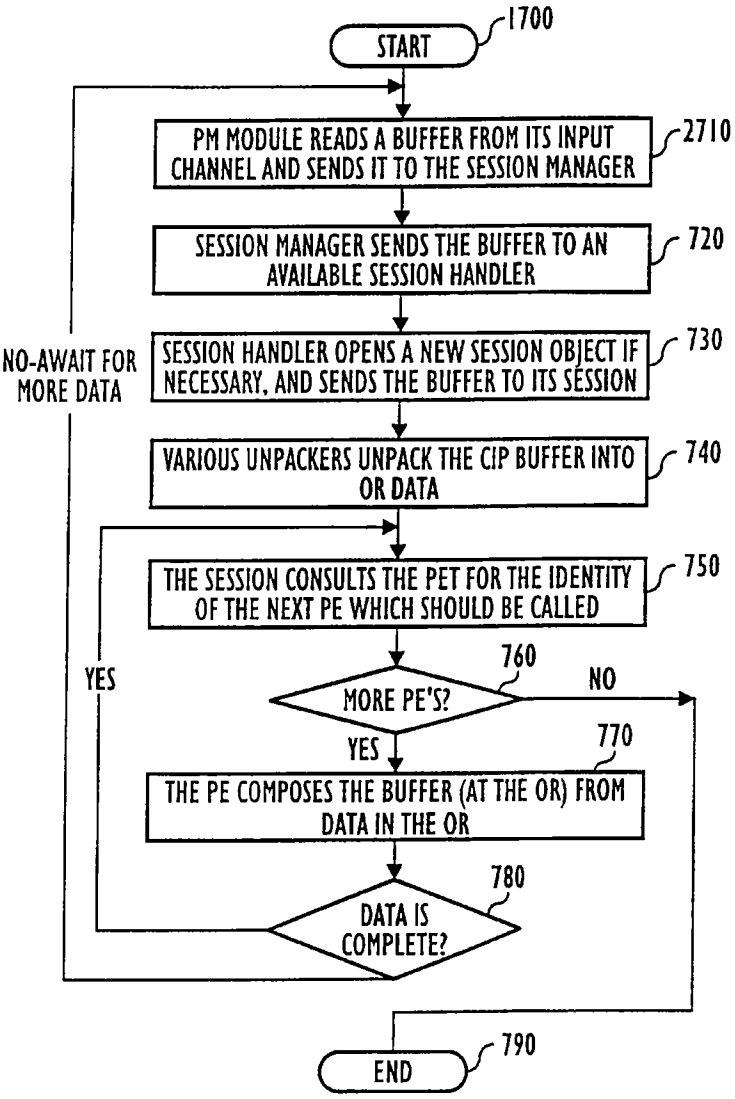

START
1700
PM MODULE READS A BUFFER FROM ITS INPUT CHANNEL AND SENDS IT TO THE SESSION MANAGER
2710
SESSION MANAGER SENDS THE BUFFER TO AN AVAILABLE SESSION HANDLER
720
SESSION HANDLER OPENS A NEW SESSION OBJECT IF NECESSARY, AND SENDS THE BUFFER TO ITS SESSION
730
NO-AWAIT FOR MORE DATA
VARIOUS UNPACKERS UNPACK THE CIP BUFFER INTO OR DATA
740
THE SESSION CONSULTS THE PET FOR THE IDENTITY OF THE NEXT PE WHICH SHOULD BE CALLED
750
760
MORE PE'S?
YES
NO
770
THE PE COMPOSES THE BUFFER (AT THE OR) FROM DATA IN THE OR
780
DATA IS COMPLETE?
YES
END
790

FIG. 17

1310

| FROM PE | TO PE | RULE | CONDITION |
|---|---|---|---|
| (start) | TCP/IP | "TRUE" | |
| TCP/IP | HTTP | Port=80 | |
| HTTP | CGI | command="POST" &&PATH="*/cgi-bin" | |
| CGI | BILL-VIEW | CGI-parameter= "bill-view-app" | |
| CGI | PAYMENT | CGI-parameter= "payment"&& user ?" guest" | |
| | | | |
| . . . | . . . | . . . | . . . |

FIG. 18

# INTERNATIONAL SEARCH REPORT

International application No.
PCT/IL98/00443

A. CLASSIFICATION OF SUBJECT MATTER
IPC(6) :G06F 12/14, 13/00, 15/00
US CL : 713/200, 201
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 713/200, 201, 202
395/200.43, 200.46, 200.54, 200.55, 200.56

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | US 5,566,326 A [HIRSCH et al] 15 October 1996, abstract, figs. 1a, 1b; col. line 40 through col. 5. line 26. | 1-21 |
| A | US 4,876,664 A [BITTORF et al] 24 October 1989, see entire document. | 1-21 |
| A | US 4,937,777 A [FLOOD et al] 26 June 1990, see entire document. | 1-21 |
| A | US 5,619,657 A [SUDAMA et al] 08 April 1997, see entire document. | 1-21 |
| A | US 5,689,708 A [REGNIER et al] 18 November 1997, see entire document. | 1-21 |

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search
13 MARCH 1999

Date of mailing of the international search report
11 MAY 1999

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer
DIEU-MINH THAI LE
Telephone No. (703) 305-9408

Form PCT/ISA/210 (second sheet)(July 1992)★